Universidade de Aveiro Departamento de Matemática
2018

Rui Daniel Soares Bettencourt

Automatização de tarefas no contexto da operação de marketing da centroProduto

Universidade de Aveiro Departamento de Matemática
2018

Rui Daniel Soares Bettencourt

Automatização de tarefas no contexto da operação de marketing da centroProduto

Relatório de estágio apresentado à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Matemática e Aplicações, realizada sob a orientação científica do Doutor António Ferreira Pereira, Professor Auxiliar do Departamento de Matemática da Universidade de Aveiro

o júri / the jury

presidente / president

Prof. Doutor Agostinho Miguel Mendes Agra
Professor Auxiliar, Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Joaquim João Estrela Ribeiro Silvestre Madeira
Professor Auxiliar, Universidade de Aveiro

Prof. Doutor António Ferreira Pereira
Professor Auxiliar, Universidade de Aveiro

agradecimentos / acknowledgements

Gostaria de agradecer à minha família, aos meus colegas, professores e funcionários da UA, em particular, o meu orientador António Pereira que me auxiliou na articulação do relatório de estágio e à equipa da centroProduto por todo o apoio prestado durante o estágio.

palavras-chave

análise de dados, normalização de dados, base de dados, concordância de sequências, extração de dados da *web*.

resumo

As tarefas apresentadas neste relatório de estágio visaram simplificar a operação de marketing da centroProduto. A centroProduto é um site de negócios digitais que visa promover marcas e empresas e estabelecer ligações comerciais entre entidades de negócios. As tarefas que os funcionários executam na centroProduto centram-se em torno de extração e análise de informações e, em seguida, aplicá-las corretamente. As tarefas que desenvolvi consistiram principalmente em criar ou melhorar programas para ajudar as atividades da empresa.

A maioria dos programas aqui apresentados tiveram o objetivo de automatizar o processamento de grandes volumes de dados. Alguns desses dados tiveram de passar por processos de tratamento e normalização antes de serem processados. Além disso, os programas tiveram de ser de fácil utilização e feitos à medida dos gostos dos utilizadores que os operam. Sempre que possível, tentei reduzir a carga de trabalho do utilizador por meio de soluções mais práticas e, ocasionalmente, automatizá-lo completamente. Quanto aos programas que interagem com ativos importantes da empresa, dediquei especial atenção aos mecanismos de prevenção e manipulação de erros.

Alguns dos temas abordados em todas as tarefas incluem: *extração de dados da web*, análise de dados, base de dados e, tangencialmente, processamento de linguagem natural e interfaces / melhoramento da experiência do utilizador.

keywords data analysis, data normalization, databases, sequence matching, web scraping.

abstract

The tasks presented in this internship report were aimed at streamlining centroProduto's marketing operation. centroProduto is a digital-business website which aims at promoting brands and companies and establishing commercial connections between business entities. The tasks employees do at centroProduto center around extraction and analysis of information and then applying it correctly. My assignments consisted mostly in creating or improving programs to aid the company's activities.

Most of programs here presented had the objective of automatizing the processing of large volumes of input data. Some of this data had to undergo treatment and normalization before becoming processable. Additionally, the programs had to be user-friendly and tailor-made to the users who operate them. Whenever possible, I tried to reduce the user's workload via more practical solutions and occasionally, automatize it completely. As for the programs which interact with important company assets, I have dedicated special attention in error prevention and handling mechanisms.

Some of the themes covered throughout the tasks include: web scraping, data analysis, databases and, tangentially, natural language processing and interfaces/user-experience.

Índice

Índice de Listagens

Índice de Tabelas

Índice de Figuras

1. Introdução

1.1 Apresentação da empresa centroProduto

A centroProduto gere e desenvolve um portal de negócios que serve como rede comercial entre empresas. O portal é especialmente direcionado a pequenas e médias empresas de forma a as ajudar a encontrar novas oportunidades de negócio, a fomentar a interação comercial entre outras empresas e a expandir o seu negócio para o estrangeiro, [1], [2], [3].

A centroProduto também desenvolve algoritmos que visam promover marcas e recomendar produtos e serviços de maneira adaptada aos interesses e necessidades dos seus clientes. O portal abrange empresas de diversos países, procurando, portanto, oferecer uma plataforma multilinguística aos seus clientes.

Para ajudar os clientes nas interações comerciais, o portal oferece funcionalidades entre as quais se destacam a criação de páginas-perfil da empresa, bem como a organização de produtos por catálogos com a informação associada. O portal permite ainda que as empresas façam controlo regional sobre o conteúdo da sua informação e os seus produtos.

O *CEO* da centroProduto, Engenheiro Pedro Sousa Rêgo, também meu orientador na empresa, é quem dirige a empresa e decide a direção das atividades desempenhadas, incluindo as tarefas descritas neste relatório de estágio. A centroProduto conta também com uma equipa (incluindo o *CEO*) de programadores e profissionais de *marketing*. A empresa, além de mim, aceita outros estagiários, entre os quais destaco uma colega de curso, Ana Cláudia Costa, que me deu suporte no plano de desenvolvimento e nos testes dos programas. No desenvolvimento das minhas tarefas, os membros da empresa, tanto programadores como profissionais de *marketing*, com capacidades que por vezes intersetam ambas as áreas, prestaram auxílio nas ocasiões em que o necessitei.

1.2 O meu papel na centroProduto

Como estagiário, as funções que desempenhei na centroProduto obedecem a um programa acordado entre a Universidade de Aveiro (mais concretamente, o Departamento de Matemática) e a centroProduto. O conteúdo do meu programa visou:

1) A introdução ao portal da centroProduto, para me familiarizar com o portal, conhecer as suas aplicações e funcionalidades. Esta fase teve por objetivo a contextualização das atividades por mim desempenhadas na empresa, bem como a participação e integração com o resto da equipa na fase de testes.
2) A execução de várias tarefas de programação orientadas aos testes, à inserção massiva de dados, à análise automática de erros, à análise estatística do desempenho da operação, à atividade de *marketing*, coleta e tratamento de informação e a outros processos que eventualmente possam surgir.
3) Definição de comportamentos esperados acerca dos programas desenvolvidos.

O meu papel nas atividades distribuídas pela empresa, em função do meu desempenho, gradualmente centrou-se cada vez mais em tarefas de programação.

As tarefas desenvolvidas na área da programação, relacionaram-se com:

- A automatização de processos de extração de informação.
- A análise e classificação de dados.
- Criação de *bots* para interação com *websites*.
- Administração de bases de dados.

A linguagem de programação usada foi, quase exclusivamente, o *Python*, com recurso a diversas bibliotecas livremente disponíveis. O *Python* destaca-se das restantes linguagens na medida em que torna o código mais leve e condensado, bem como exige que este seja devidamente indentado, o que resulta num código fácil de ler. Embora tenha tido experiência com o *Python* durante o meu percurso académico, ao longo do estágio foi necessário aprofundar ainda mais os conhecimentos no uso da linguagem e das suas bibliotecas.

1.3 Práticas gerais na centroProduto

Plano de desenvolvimento

Os programas desenvolvidos na empresa centroProduto seguem um plano de desenvolvimento. O plano de desenvolvimento é uma folha de cálculo que detalha as especificações de um programa, como por exemplo: o controlo de qualidade dos dados de entrada/saída, o comportamento esperado, ou as funcionalidades. O plano de desenvolvimento é tipicamente escrito por membros que não o programador.

O controlo de qualidade dos dados detalha o tipo e características dos dados, a que estes devem obedecer. O comportamento esperado de um programa define a forma como este deve responder face a diversas situações que surgem. As funcionalidades de um programa referem-se às ações que um programa consegue executar.

Testes

Uma das exigências da empresa dita que os programas desenvolvidos requerem a realização de testes antes de passarem à aplicação real. A fase de testes de um programa é uma tarefa que exige método, paciência e atenção. Geralmente os testes implicam verificar numerosas situações que são combinações de estados. Os testes sobre os programas podem ser ainda mais morosos quando os dados de entrada/saída vêm em formato de ficheiro ou com estruturas complexas.

Por vezes foi possível agilizar os testes via programação gerando dados com várias combinações de estados ou usando programas auxiliares que permitem inserir e ler os dados mais complexos tornando os testes mais rápidos e práticos de executar. No entanto, nem sempre é possível usar a programação para os testes, nessas situações a única alternativa é inserir os dados manualmente e analisar cada situação.

Outra exigência da empresa dita que os testes devem ser feitos por membros diferentes de quem desenvolve o programa. De acordo com a perspetiva da empresa, o programador pode estar demasiado centrado num método de resolução de um problema ficando alheio a alternativas melhores. De forma análoga, um método pode conseguir tratar erros

que a outro escapam. Na secção 1.4 detalho os primeiros testes que realizei no portal centroProduto.

Documentação

A documentação de um programa consiste num documento de texto que detalha a informação acerca do seu funcionamento como por exemplo: os ficheiros que recebe e devolve, a maneira como deve ser utilizado, a explicação de algoritmos mais complexos e as bibliotecas dependentes.

Versionamento dos programas

Os programas desenvolvidos têm uma versão assinalada. A versão costuma ser um número incremental. Sempre que um programa é alterado e submetido para testes, ou guardado nos servidores, deve receber uma nova versão.

As alterações que os programas recebem são sempre descritas no cabeçalho do programa em comentário. Os projetos de maior dimensão que envolvem vários membros têm documentos dedicados às alterações, indicando quem as deve implementar e quando as implementou.

Nos capítulos que se seguem, apresento as tarefas realizadas na empresa por ordem cronológica. No capítulo 7 descrevo algumas tarefas realizadas que, ou têm aspetos comuns às restantes tarefas, ou não requerem um capítulo dedicado. Na conclusão deste documento, explico como as tarefas se englobam na escala maior da operação de *marketing* da centroProduto.

1.4 Testes de rotas no portal da centroProduto

Esta foi a primeira tarefa que desempenhei na centroProduto. O objetivo consistiu em realizar testes de casos de uso no portal centroProduto. Esta tarefa serviu também como introdução ao portal da centroProduto e à maneira como a equipa realiza os testes na empresa. Adicionalmente, também serviu para me integrar com alguns membros da equipa.

O termo “caso de uso” provém da engenharia de *software*, que serve para designar uma funcionalidade que um sistema oferece. Na empresa, é mais frequente ouvir o termo “rota” para designar os casos de uso, isto deve-se ao facto de que as rotas por vezes são mais extensas em relação ao número de passos executados do que as simples ações que os casos de uso tipicamente detalham.

Os vários casos de uso a serem testados vinham descritos num ficheiro folha de cálculo, cada linha indicando um caso de uso e algumas condições que tinham de ser cumpridas ao executar os testes.

O portal centroProduto possui várias instâncias, entre elas:

- A instância real, que é o portal disponível ao público.
- A instância de desenvolvedor (*developer*), que serve para desenvolver funcionalidades e realizar testes sobre as mesmas.

Como o portal real é uma plataforma de negócios, com utilizadores e transações reais, é necessário existir um ambiente isolado que permita interagir sobre as várias entidades e eventos sem que os testes tenham consequências nas atividades reais.

Após me ter sido atribuída uma conta na instância de desenvolvedor do portal, foi-me dada a liberdade para explorar livremente, durante algumas horas, as funcionalidades do portal, como se fosse um utilizador com empresas registadas. Só depois de estar familiarizado com o portal, é que comecei a testar os vários casos de uso.

Estes testes têm de ser realizados de cada vez que uma nova atualização é desenvolvida e antes de ser lançado na instância real.

Ao mesmo tempo que executava os testes, a empresa também aproveitou a minha inexperiência destacando um dos membros da equipa para observar o meu comportamento enquanto interagia com o portal pela primeira vez. Esta é uma prática comum com os estagiários e novos membros da equipa na empresa, pois ajuda a perceber de perto as tendências dominantes e as limitações sentidas pelos utilizadores reais do portal, os quais são maioritariamente inexperientes.

No meu caso, tive alguma dificuldade (percetível) na maneira como tentava aceder aos menus. Tipicamente, os menus estão localizados em zonas no cabeçalho ou à esquerda

do ecrã, e é para estas áreas que a minha intuição me levava. Também senti alguma dificuldade em encontrar determinados botões, mesmo à minha frente quando não estavam no sítio que esperava estarem.

Numa outra situação, foi-me perguntado porque é que estava a carregar no teclado quando tentava selecionar opções em listas. Esperava, como é usual, que a posição saltasse para as palavras das listas correspondente às letras das teclas pressionadas, como acontece em muitos *websites*.

Os testes que realizei levaram cerca de 12 horas a serem completados. Isto deu-me uma ideia concreta do imenso trabalho e da morosidade do processo de realizar testes.

2. Web Scraping

Objetivo: Melhorar/otimizar um programa que extrai informação relativa a empresas.

Esta foi a primeira tarefa de programação que desenvolvi na empresa. Foi-me dado um programa em mãos que extrai informação acerca de empresas a partir de *websites* para o melhorar. À extração informação de *websites* a partir de *bots*, como um *webdriver* controlado por um programa, chama-se *web scraping*.

Os *websites* pelos quais a informação é obtida autorizam a extração dos dados das empresas. Os programas apresentados neste capítulo apenas tornam mais eficiente um processo que pode ser feito manualmente por qualquer pessoa através dos *websites*.

Procedimento 1: Programa original
Para cada nome legal de uma lista de empresas: ▪ Através de um *webdriver*, fazer a procura pelo nome da empresa num *website*. ▪ Esperar que o utilizador selecione uma *URL* entre os resultados da procura. ▪ No caso em que uma *URL* pertencente a uma empresa foi aberta o programa extrai a informação da empresa.

O programa original, ver procedimento 1, percorre uma lista com nomes de empresas, em uma folha de cálculo e serve-se de um *webdriver* para simular um utilizador a interagir com os *websites*, extraindo a informação das suas páginas.

As várias bibliotecas, em *Python*, que este programa usa são:

- O *openpyxl*, [4], que serve para operar sobre folhas cálculo.
- O *BeautifulSoup4*, [5], que serve para percorrer o código fonte dos *websites*.
- O *Selenium*, [6], que em conjunto com um *webdriver* permite interagir com elementos das páginas dos *websites*.

Estas bibliotecas, bem como o conceito de *web scraping*, foram novos para mim. Tive de aprender a usar estas três bibliotecas rapidamente para conseguir mostrar resultados ao

fim de três dias de estágio. Foi também interessante aprender como um programa podia interagir com *websites* e extrair informação de páginas. A habilidade de construir programas que fazem *web scraping* passou a ser bastante útil para tarefas realizadas na empresa, direta e indiretamente.

2.1 Melhoramentos

Após ter dominado minimamente as bibliotecas e ter analisado bem o programa, reparei que o programa era desnecessariamente lento, executando passos supérfluos e dependendo da decisão de um utilizador.

O programa original abre um *webdriver* (do *Google Chrome*), acede à página inicial, escrevendo o nome da empresa na caixa de procura, executando a procura, e finalmente espera pela interação do utilizador que tem de escolher um resultado da procura no *website*. Após a escolha de um resultado, o programa extrai a informação da empresa e guarda-a num ficheiro de saída.

Facilmente se observa, após uma pesquisa nos *websites*, que as *URL* das pesquisas contêm um padrão composto por um prefixo e por um sufixo constantes, sendo a parte variante dependente do que é pesquisado.

A parte variante é uma reconstrução do nome da empresa (como é inserida na procura), sem a maior parte dos caracteres não alfanuméricos, e espaçados pelo carácter '+' (mais), ver exemplo 1.

O primeiro melhoramento, mais imediato, foi simplesmente gerar a *URL* de pesquisa com o mesmo padrão e aceder diretamente à página dos resultados. Desta maneira, eliminei um carregamento de uma página desnecessário.

Como interessava procurar por empresas apenas, nos *websites* que permitem filtrar por empresas (bem como marcas, ou outros critérios), se fosse possível inserir esses parâmetros via *URL*, então ajustava a *URL* para filtrar instantaneamente os resultados.

Exemplo 1

Aqui mostro como uma procura do nome “Centro Produto” num *website*, aqui designado por “website_a”, é executada com a seguinte *URL* de pesquisa:

https://www.website_a.com/pesquisa/?q=centro+produto&tipo=empresas

As componentes desta *URL* são:

- Prefixo: https://www.website_a.com/pesquisa/?q=
- Parte variante (da procura): centro+produto
- Sufixo: &tipo=empresas

A parte variante converteu o espaço no carácter ‘+’. O sufixo ‘&tipo=empresas’ filtra imediatamente por empresas.

O segundo melhoramento consistiu em mitigar a utilização do *webdriver* para os *websites* que não o necessitassem. Existem *websites* que requerem a interação do utilizador, ou que precisam de ser carregados num *web browser* para dispor da informação completa. Nestes casos, não se pode mitigar a utilização de um *webdriver* por completo.

Para os *websites* que não requerem um *webdriver*, o qual demora alguns segundos a carregar a informação da página, passei a usar uma biblioteca em *Python* chamada *requests*, [7], que permite descarregar os *websites* mais rapidamente.

O terceiro melhoramento, mais significativo, consistiu em eliminar a necessidade de um utilizador acompanhar o processo de extração, por completo. Em vez de o utilizador estar presente durante a extração e manualmente escolher o melhor resultado da pesquisa e abrir a *URL*, fiz com que o programa decidisse o melhor resultado por si só, em frações de segundos.

Na página de resultados tinha os nomes das empresas resultantes da procura e queria achar qual desses resultados melhor se assemelhava ao nome da empresa que foi procurada. Este é um problema de concordância de sequências (*sequence matching*), o qual vou explicar melhor na próxima secção.

2.2 Concordância de sequências

Existem ferramentas que conseguem medir o quão semelhantes duas *strings* são uma da outra. O *Python* possui uma biblioteca para este fim chamada *difflib*, [8]. A biblioteca *difflib* oferece algumas ferramentas para comparar sequências, uma delas, o *SequenceMatcher*, compara a semelhança entre duas *strings*, devolvendo um rácio entre 0.0 e 1.0. Quanto mais perto for o rácio de 1.0 mais semelhantes as *strings* são.

Algoritmo de medição de similaridade entre *strings*

O algoritmo *SequenceMatcher* é uma versão melhorada de um algoritmo publicado em 1983 por John W. Ratcliff e John A. Obershelp, denominado por "*Gestalt Pattern Matching*, [9]. *Gestalt* é uma palavra Alemã que significa, literalmente: "forma" ou "feitio"; no contexto mais abrangente significa que o todo tem maior significado do que a soma das partes individuais.

De acordo com [9], este algoritmo influenciou bastante o *software* educacional, o qual, até então, apenas podia ou receber respostas por escolha múltipla, ou receber respostas escritas, mas que estivessem corretas carácter por carácter. Com este algoritmo, foi possível aceitar respostas com alguma margem para erros ortográficos, mas que de outra forma, estariam certas se fossem suficientemente semelhantes à resposta correta.

Ainda segundo [9], para além do *software* educacional, este algoritmo também teve aplicações em outras áreas, permitindo sugerir correções sobre erros ortográficos em programas de edição de texto e compiladores, apurar registos provavelmente repetidos em bases de dados, melhorar as mecânicas dos videojogos do género de aventuras de texto.

Explicando de forma simples, este algoritmo procura a *substring* comum mais longa presente nas duas *strings*, dadas como parâmetros de entrada. Esta *substring* é considerada como uma âncora, onde o algoritmo volta depois a procurar a *substring* comum mais longa para as partes direitas e para as partes esquerdas restantes das *strings*. O algoritmo repete este processo até que as *substrings* sejam ou completamente diferentes, ou vazias, [10].

Como as *strings* são unidimensionais, este algoritmo devolve um valor que pode ser interpretado como uma percentagem do quanto duas *strings* são semelhantes, ver o exemplo 2.

Exemplo 2: Aplicação do algoritmo

s1 e s2 duas *strings*. O algoritmo vai ver o quanto semelhante a *string* s2 é de s1.

s1	A	L	G	O	R	I	T	M	O	
s2	A	G	O	R	A		M	O	V	E

Na primeira iteração, a *substring* comum mais longa que o algoritmo acha é “GOR”, preenchida a verde.

s1	A	L	G	O	R	I	T	M	O	
s2	A	G	O	R	A		M	O	V	E

Ainda existem *substrings* não vazias à esquerda e direita de “GOR”. Na próxima iteração, em ambas as *strings*, o algoritmo procura pela *substring* comum mais longa entre as *substrings* restantes à direita: “ITMO” de s1 e “A MOVE” de s2; e à esquerda: “AL” de s1 e “A” de s2.

As *substrings* comuns mais longas encontradas nesta iteração são preenchidas a laranja e a azul.

s1	A	L	G	O	R	I	T	M	O	
s2	A	G	O	R	A		M	O	V	E

Depois, o algoritmo ainda continua, mas:

- à esquerda da *substring* comum “A”, apenas tem *substrings* vazias, por isso o algoritmo termina a procura pelo lado direito.
- à direita da *substring* comum “A”, apenas tem a *substring* “L” de s1 e a *substring* vazia de s2, por isso o algoritmo também termina a procura pelo lado esquerdo.
- à esquerda da *substring* “MO”, existem a *substrings* “IT” de s1 e “A ” de s2, mas são absolutamente diferentes, por isso o algoritmo termina a procura pelo lado esquerdo.

- à direita da *substring* “MO”, tem a *substring* vazia de s1 e “VE” de s2, por isso o algoritmo termina a procura pelo lado direito, e não há mais por onde procurar.

As *substrings* comuns mais longas encontradas são: “A”, “GOR” e “MO”.

Este algoritmo também permite o cálculo de uma classificação de similaridade.

Esta classificação é obtida pelo dobro soma do número de elementos encontrados (neste caso, caracteres, 2*6=12, a dividir pelo número total de elementos (neste caso, a soma do comprimento) das duas *strings* originais, 19.

Portanto a similaridade entre s1 e s2 é 12/19 ~ 0.632...

Replicação do algoritmo “Gestalt Pattern Matching”

Para exemplificar e analisar em mais detalhe, implementei uma versão recursiva do algoritmo, [11], através da listagem 1.

Para representar as *substrings* e os seus intervalos, crio a classe SubString(), que é composta pelos atributos:

- *string*: que é uma *string* regular, contém os caracteres das (*sub*)*strings*.
- start: que indica a posição, relativa à *string* original, onde começa.
- end: que indica a posição, relativa à *string* original, onde acaba.

Se a (*sub*)*string* desta classe for inicializada sem um parâmetro de posição de início (start), então assume o valor 0 e é considerada uma *string* original. A posição de fim é atribuída automaticamente a partir do comprimento da (*sub*)*string*, mais a posição inicial. O método *length()* é supérfluo, mas deixo assim por uma questão de formalidade.

Listagem 1: Gestalt Pattern Matching, por Ratcliff e Obershelp

```
class SubString():
    def __init__(self, string, **kwargs):
        self.string = string

        if 'start' in kwargs: start = kwargs['start']
```

```
        else: start = 0
        self.start = start

        self.end = start+len(string)

    def length(self):
        return self.end-self.start
```

A função principal - *ratcliff_obershelp()* - recebe dois parâmetros do tipo *string* e converte-os na classe *SubString()*. A função instancia também uma lista (matches) que guarda as *substrings* comuns mais longas. Depois chama a função recursiva para ambas *strings* (*SubString()*) que é onde o algoritmo começa.

Na função recursiva, é encontrada a *substring* comum mais longa. Se esta *substring* não for vazia, o algoritmo adiciona-a à lista matches e continua.

A seguir, o algoritmo separa as restantes partes à esquerda e à direita da *substring* encontrada. O algoritmo volta a tentar achar, recursivamente, a *substring* mais longa entre as restantes partes.

Listagem 2: Gestalt Pattern Matching, por Ratcliff e Obershelp

```
def ratcliff_obershelp(a, b):

    def recursion(a, b):

        lcs_a, lcs_b = find_longest_common_substring(a, b)
        if lcs_a.length() == 0 or lcs_b.length() == 0: return

        matches.append(lcs_a)

        left_of_a  = SubString(a.string[0:(lcs_a.start-a.start)],
start=a.start)
        right_of_a = SubString(a.string[(lcs_a.start-
a.start+lcs_a.length()):a.length()], start=lcs_a.end)
        left_of_b  = SubString(b.string[0:(lcs_b.start-b.start)],
start=b.start)
        right_of_b = SubString(b.string[(lcs_b.start-
b.start+lcs_b.length()):b.length()], start=lcs_b.end)

        recursion(left_of_a, left_of_b)
        recursion(right_of_a, right_of_b)

    #SETUP
    a = SubString(a)
```

```
    b = SubString(b)

    #MATCHES
    matches = []
    recursion(a, b)

    #RATIO
    score = 0
    for match in matches: score += match.length()
    ratio = score*2/(a.length()+b.length())

    return ratio, matches
```

A função que acha a *substring* de comprimento mais longo, percorre todas as combinações de posições por onde se começa a testar uma carreira de caracteres iguais.

A partir a da posição *i* na *string_a* e da posição *j* na *string_b*, testa-se se a *string_a* na posição *i+k* é igual à *string_b* na posição *j+k*. Se sim *k* é incrementado, e continua a testar enquanto forem iguais e estiverem dentro do comprimento de ambas *strings*.

Se para uma determinada combinação de posições iniciais, *i* e *j*, existir um *k* maior que a *substring* encontrada até então, esta é selecionada como a *substring* comum mais longa encontrada até este momento.

Listagem 3: Gestalt Pattern Matching, por Ratcliff e Obershelp

```
def find_longest_common_substring(a, b):

    lcs_a = SubString('', start=a.start+a.length())
    lcs_b = SubString('', start=b.start+b.length())

    for i in range(a.length()):
        for j in range(b.length()):
            k = 0
            while i+k < a.length() and j+k < b.length() and a.string[i+k] ==
b.string[j+k]: k += 1

            if k > lcs_a.length():

                lcs_a=SubString(a.string[i:i+k], start=i+a.start)
                lcs_b=SubString(b.string[j:j+k], start=j+b.start)

    return lcs_a, lcs_b
```

Finalmente, a função calcula o rácio de similaridade que é dado pelo dobro da soma do comprimento das *substrings* na lista match, divida pela soma do comprimento das *strings* iniciais.

Aplicação do *SequenceMatcher*

O seguinte código em *Python*, listagem 4, importa a biblioteca *difflib*, instancia duas *strings*: “ABCxDEyF” e “ABCDEzFw” e inicializa o objeto da sequência.

Listagem 4: Aplicação do *SequenceMatcher*

```
import difflib

string_a = "ABCxDEyF"
string_b = "ABCDEzFw"

sequence = difflib.SequenceMatcher(a=string_a, b=string_b)
```

O método *.get_matching_blocks()* devolve um terno que descreve o intervalo de posições em que a *string_b* condiz com a *string_a* e o seu comprimento (*size*). O último terno é apenas um terno fictício da *substring* vazia. O método *.get_opcodes()* permite obter a uma lista de transformação da *string_a* para ser transformada na *string_b*.

Convém observar que esta função não é comutativa. O que o algoritmo faz é tentar encontrar as *substrings* da *string_b* que mais estiverem contidas na *string_a*. A mesma função com os parâmetros revertidos pode resultar em *substrings* diferentes.

Listagem 5: Aplicação do *SequenceMatcher*

```
for match in sequence.get_matching_blocks():
    print(match)
```

```
Match(a=0, b=0, size=3)
Match(a=4, b=3, size=2)
Match(a=7, b=6, size=1)
Match(a=8, b=8, size=0)
```

Listagem 6: Aplicação do *SequenceMatcher*

```
for tag, i1, i2, j1, j2 in sequence.get_opcodes():
    print ("%7s a[%d:%d] (%s) b[%d:%d] (%s)" % (tag, i1, i2,
           string_a[i1:i2], j1, j2, string_b[j1:j2]))
```

```
  equal a[0:3] (ABC) b[0:3] (ABC)
 delete a[3:4] (x)   b[3:3] ()
  equal a[4:6] (DE)  b[3:5] (DE)
replace a[6:7] (y)   b[5:6] (z)
  equal a[7:8] (F)   b[6:7] (F)
 insert a[8:8] ()    b[7:8] (w)
```

O código da listagem 6 foi retirado da referência [8].

Para transformar a *string_a* na *string_b*, os seguintes passos têm de ser executados: a *substring* "ABC" é comum a ambas as *strings* por isso não se altera nada, "x" é apagado da *string_a*, "DE" é comum, "y" é substituído por "z", "F" é comum e "z" é inserido na *string_b*.

operação	igual	apaga	igual	substitui	igual	insere
string_a	ABC	x	DE	y	F	
posições	[0:3]	[3:4]	[4:6]	[6:7]	[7:8]	
string_b	ABC		DE	z	F	w
posições	[0:3]		[3:5]	[5:6]	[6:7]	[7:8]

Tabela 1: Operações de transformação entre *strings*

O método *.ratio()* devolve um rácio do quão a *string_b* é semelhante à *string_a*. Este rácio é calculado pela fórmula $2 * M/T$, onde *M* é o número de elementos (caracteres) que são iguais e T é o número total de elementos (a soma do comprimento da *string_a* e *string_b*). Naturalmente, duas *strings* iguais dão um rácio de 1.0 e duas *strings* absolutamente diferentes dão um rácio de 0.0.

Listagem 7: Aplicação do *SequenceMatcher*

```
print("sequence.ratio() =", sequence.ratio())
M = 0
for a, b, size in sequence.get_matching_blocks():
    M += size
```

```
T = len(string_a) + len(string_b)

print("Manual ratio calculation:", 2*M/T)
```

```
sequence.ratio() = 0.75
Manual ratio calculation: 0.75
```

Aplicação prática

No programa original, removi a parte do código encarregue de esperar pela interação do utilizador e substituí por uma função que escolhe qual o nome da empresa, nos resultados, mais parecida com a empresa que procurei, usando a biblioteca *difflib* para comparar e obter o resultado com o melhor rácio.

Antes de comparar as *strings*, normalizava os nomes das empresas/resultados:

- Convertia tudo em minúsculas.
- Retirava a acentuação.
- Retirava caracteres não alfanuméricos.
- Retirava espaçamentos com mais de um carácter.
- Retirava também a responsabilidade legal das empresas (“Limitada”, “Sociedade Anónima” e as suas variações).

Este passo foi bastante importante na medida em que impedia que diferenças entre as *strings* comparadas, que não têm qualquer importância (tais como pontos, vírgulas e traços), afetassem o valor do rácio de similaridade.

Existiam resultados cujas *strings* eram mais longas do que o nome da empresa pela qual o programa procurava. Por vezes os nomes logo no início eram iguais, mas como o resto da *string* fazia com que o rácio de similaridade diminuísse, fiz uma pequena personalização no cálculo da similaridade.

Acrescentei a condição: quando o resultado a ser comparado é maior do que 1.1 vezes o tamanho da *string* da empresa que se procura, a *string* do resultado é truncada e deduz-se 0.1 (10%) no rácio. Desta forma foi possível melhorar ligeiramente a extração de informação. No entanto, esta condição era apenas útil para nomes de empresas únicos, empresas com nomes próprios dão resultados incorretos com frequência.

Tomei a decisão de aceitar resultados com rácios de similaridade iguais ou superiores a 0.85 (85%). A decisão desta fasquia foi baseada nos testes realizados. 85% de similaridade já é uma condição exigente.

Para dar uma ideia sobre a confiança dos dados, guardei o rácio da similaridade, o nome/designação do melhor resultado selecionado, a normalização do nome, mais a *URL* da página de onde foi extraída a informação no ficheiro de saída.

Exemplo 3: Utilização do rácio de similaridade para apurar os melhores resultados

Nome da empresa para procurar no ficheiro de entrada: “AMBICARE INDUSTRIAL - TRATAMENTO DE RESÍDUOS, SA”

Nome simplificado da empresa: “ambicare industrial tratamento de residuos”

ambicare industrial tratamento de residuos	1.0
gestzende gestao e tratamento de residuos	0.67
recinovas tratamento de sucatas unipessoal	0.55
sisav sistema integrado de tratamento e eliminacao de residuos	0.42
aluline portugal drenagem e tratamento de aguas unipessoal	0.4
resaqua projectos e gestao industrial	0.33
socipole sociedade industrial de perfumes oleos e limpezas	0.38

O programa recolhe o resultado que tem o melhor rácio de similaridade (“ambicare industrial tratamento de residuos” com 1.0).

2.3 Web Scraping

Para extrair a informação o programa serve-se de dois *websites*, doravante identificados por *Website_A* (www.website_a.com) e *Website_B* (www.website_b.pt). Estes *websites* contêm informação sobre empresas como por exemplo: o NIF (número de identificação

fiscal), o CAE (atividade) e o CEP (código postal), entre outras coisas, que interessa à empresa obter para completar a informação em falta na sua base de dados.

Para o programa conseguir extrair a informação de *websites*, este usa algumas bibliotecas em *Python*.

A biblioteca *requests* oferece uma instrução que permite obter o código fonte de uma página, através de uma *URL* como parâmetro de entrada.

A biblioteca *Selenium* permite controlar um webdriver, simulando o uso do *website* por um utilizador, para interagir com os elementos da página de modo a obter a informação de outro modo inacessível. O *Selenium* oferece instruções para interagir com elementos das páginas (botões, caixas de textos), permite simular o carregamento de teclas ou movimentos do rato, consegue introduzir informação em caixas de texto e com algumas técnicas de programação, é possível simular o atraso e aleatoriedade naturais de movimentos na interação que um utilizador (humano) tem.

O *requests* e o *Selenium* são usadas para descarregar o código fonte das páginas. O *Selenium* destaca-se do *requests* por conseguir extrair mais informação e pela sua capacidade de interação com os elementos da página. O *requests*, por sua vez, demora menos tempo a retornar informação. No entanto, existem páginas que não carregam toda a informação a menos que esta seja visualizada por um *web browser*, nestes casos, o *Selenium* é necessário.

A biblioteca *BeautifulSoup4* permite percorrer a informação no código fonte das páginas de forma prática. É com esta biblioteca que se extrai a informação das empresas a partir do código fonte.

O *website Website_A* permite que se extraia informação apenas com a biblioteca *requests*, prescindindo do uso de um webdriver. Isto torna este *website* a fonte mais rápida para a extração de informação. O *Website_B*, por outro lado, requer o uso de um webdriver, mas por vezes contém informação que o *Website_A* não contém.

A ideia para otimizar a extração da informação de empresas é então procurar primeiro no *Website_A* e depois, se necessário, procurar no *Website_B*.

As procuras são realizadas consoante a informação em falta no ficheiro de entrada. Se apenas existir o nome a da empresa, o programa procura pelo nome. Se o NIF estiver preenchido, então o programa faz a procura pelo NIF, seja esta uma procura literal ou tentando aceder à página diretamente pelo código do NIF na *URL* (que é possível em alguns sites, como é o caso com o *Website_B*). Procurar a página da empresa pelo NIF garante um resultado correto.

robots.txt

Os *websites* contêm um ficheiro robots.txt, convencionalmente colocado na raíz do *website*, que permite, ou proíbe que *robots* da web - por exemplo, um programa que faz procuras através de um *webdriver* - possam interagir com o *website*, [12].

Este ficheiro pode ser acedido, tipicamente, ao adicionar "/robots.txt" no fim do link raiz de um *website*. Por exemplo, para o *website* mais usado na extração de informação desta tarefa, www.website_a.com, o ficheiro pode ser encontrado em www.website_a.com/robots.

A informação contida neste ficheiro dita que *robots* são permitidos ou proibidos, que partes do *website* podem ser acedidas ou bloqueadas.

Exemplo 4: robots.txt que permite qualquer tipo de utilização

```
User-agent: *
Allow: /
```

A configuração mais vantajosa para quem pretender usar um *webdriver* e aceder livremente a um *website*, é a que permite que o seu (ou qualquer) robot tenha acesso a qualquer parte do *website*. No exemplo 4, a primeira linha dita que isto se aplica a todos os *robots* (*User-agent*). A segunda linha explicita que têm acesso a qualquer parte (diretório) do *website*.

Nos exemplos dos *websites* usados para extração, estes têm a mesma configuração apresentada acima, mas é possível que alguns *websites* apresentem configurações diferentes. Alguns podem simplesmente bloquear alguns ou quaisquer *robots* a algumas ou

todas as partes do *website*. Embora seja possível tentar "enganar" os *websites* com alguns truques (como alterar a localização do *webdriver* via *proxies* ou simular a aleatoriedade e a demora de movimentos de um utilizador humano), não existe, por parte da empresa, a intenção de agir de má fé.

Convém, no entanto, ter a noção que, mesmo com algumas restrições, é possível montar um programa de extração de informação que apenas tome partido do que lhe é permitido via as instruções no ficheiro, sem necessidade de perder tempo com truques ou com um código que resulta num extrator que é apanhado e tornado obsoleto.

2.4 Desempenho

No ficheiro de testes, a partir de uma lista com 283 empresas, que já havia sido processada pelo programa original, o programa achou 260 empresas (92%), ver tabela 2. Das empresas encontradas, 200 delas (77%) tinham um rácio de similaridade de 1.0, 15 (6%) tinham um rácio acima da fasquia de 0.85, mas inferior a 1.0 e as restantes 45 (17%) eram resultados truncados, ver tabela 3.

Em um ficheiro com uma lista de 85801 empresas, o programa achou resultados para 78887 dessas empresas (coincidentemente, 92%), ver tabela 4. Dos 78887 resultados encontrados, 75426 (96%) tinham um rácio de similaridade 1.0, 1895 (2%) tinham um rácio acima da fasquia de 0.85, mas inferiores a 1.0 e os restantes 1566 (2%) eram resultados truncados, ver tabela 5.

É difícil dizer ao certo, sem fazer a análise caso a caso, se todos os resultados truncados estão corretos. Os resultados truncados funcionam bem para empresas com nomes únicos, mas quando os nomes são composições de nomes próprios ou palavras ordinárias, os resultados têm uma grande tendência a serem incorretos por causa de serem comuns no universo das empresas existentes.

Convém também notar que os resultados são favoráveis em grande parte devido ao facto de que as empresas, em ambos os casos, eram portuguesas e os *websites* utilizados continham a informação destas empresas.

A maior parte dos resultados tinham similaridade 1, isto deve-se principalmente ao processo de normalização das *strings* que ajudou imenso na equalização das *strings*. Mesmo que a função de similaridade não fosse aplicada, este programa ainda teria obtido bons resultados.

Nestas estatísticas não são incluídas empresas cuja procura foi feita pelo NIF. Mesmo assumindo que a informação das empresas com rácio inferior a 1.0 é incorreta, o programa achou, no caso do ficheiro com mais de 85000 entradas, cerca de 88% de resultados com similaridade 1.0.

O programa é também muito mais rápido do que o original, que dependia da decisão de um utilizador. A acurácia da informação obtida é extremamente boa relativamente à atenção e tempo que um utilizador teria que aplicar no processo original. Adicionalmente, o programa já não necessita sequer de um utilizador, poupando mão de obra à empresa.

Por estes motivos, considero que as alterações realizadas tenham tido extremo sucesso na obtenção rápida e automática dos dados.

Resultados	**Contagem**	**%**
Encontrados	260	91.9
Não encontrados	23	8.1
Total	283	100

Tabela 2: Resultados da lista de empresas no ficheiro de testes

Similaridade dos resultados encontrados	**Contagem**	**%**
1	200	76.9
[0.85, 1[	15	5.8
Truncados	45	17.3

Tabela 3: Similaridade dos resultados encontrados no ficheiro de testes

Resultados	Contagem	%
Encontrados	78887	91.9
Não encontrados	6914	8.1
Total	85801	100

Tabela 4: Resultados da lista de empresas

Similaridade dos resultados encontrados	Contagem	%
1	75426	95.6
[0.85, 1[	1895	2.4
Truncados	1566	2

Tabela 5: Similaridade dos resultados encontrados na lista de empresas

3. Inserção Rápida

Objetivo: Desenvolver um módulo que recebe informação estruturada, a analisa e devolve um dicionário-resposta com o conteúdo reestruturado ou com uma lista de erros.

O acréscimo do número de utilizadores do portal da centroProduto, em consequência da operação de *marketing*, aumenta o tempo despendido no apoio ao cliente. Muitos dos utilizadores que visitam o portal pela primeira vez, com o intuito de se inscreverem e preencher informação sobre a sua empresa deparam-se com algumas dificuldades. Grande parte das dificuldades com que os novos utilizadores se deparam relaciona-se com a sua inexperiência perante as tecnologias modernas. Com alguma frequência, a centroProduto recebe pedidos de ajuda para completar processos de inscrição no portal.

Para dar resposta a este problema, a empresa desenvolveu uma nova funcionalidade denominada *FASTINSERT()*, inserção rápida, que permite que contas administrativas do portal centroProduto possam criar utilizadores e preencher dados sobre empresas de forma mais prática. Uma conta administrativa do portal é uma conta com privilégios específicos, atribuída apenas aos funcionários da empresa.

Neste capítulo apresento um módulo que desenvolvi que se integra na funcionalidade *FASTINSERT()*, cuja função é receber dados de entrada, provenientes de um *email*, interpretar e analisar o seu conteúdo e finalmente construir um dicionário de resposta com os dados reestruturados. Este módulo está organizado nas três fases indicadas no procedimento 2.

Procedimento 2
▪ O processador de linguagem interpreta os dados de entrada. ▪ A análise de requisitos verifica se os dados necessários estão presentes. ▪ A validação dos dados faz o controlo de qualidade dos dados.

3.1 Configuração da equipa

Nesta tarefa trabalhei em conjunto com alguns membros da centroProduto, sendo por estes auxiliado e supervisionado. Sendo a primeira tarefa com aplicação direta no portal, tentei ao máximo fazer um código claro, organizado e sem erros.

A parte mais complexa deste módulo é a fase de processamento de linguagem. A fase de análise de requisitos, essencialmente, verifica se os dados estão presentes para executar determinadas operações. Para a fase de validação de dados, foi criado um conjunto de verificações simples sobre os dados, mas a restantes verificações, mais complexas, foram realizadas por funções pertencentes à biblioteca de *software* da empresa. Por este motivo, dou mais relevo à descrição da fase de processamento de linguagem.

3.2 Processador de linguagem

O módulo recebe o conteúdo do *email*, em texto, como parâmetro de entrada. O texto de entrada contém dados organizados numa estrutura *XML*. Os dados que o processador de linguagem trata estão estruturados em elementos delimitados por *tags* específicas, como por exemplo, <email_content>.

Um elemento é um terno ordenado composto por:

- Uma *tag* de abertura, que define o início do elemento, com um determinado nome encapsulado pelos caracteres '<' e '>'.
- Um conteúdo.
- Uma *tag* de fecho, que define o fim do elemento, com o mesmo nome que a *tag* de início encapsulado pelos caracteres '</' e '>'.

O exemplo 5 mostra como os dados que o processador de linguagem trata, são delimitados pelo elemento <email_content>. Os dados neste exemplo estão formatados por uma questão de apresentação, mas na prática podem aparecer condensados numa linha.

Elementos que apenas contêm texto dizem-se ter conteúdo simples. Elementos que contém outros elementos dizem-se ter conteúdo complexo, [13]. Assim, o elemento <email_content> contém conteúdo complexo enquanto a maioria dos seus subelementos contêm conteúdo simples. Existem apenas poucas exceções onde os subelementos de

<email_content> contêm conteúdo complexo. No exemplo 5, o elemento <item> contém conteúdo complexo.

Exemplo 5: Conteúdo de entrada

```
<email_content>

<email.subject> ... </email.subject>

<campaign.id> ... </campaign.id>
<operations> ... </operations>

<user.first_name> ... </user.first_name>
<user.last_name> ... </user.last_name>
<user.email> ... </user.email>
<user.language> ... </user.language>
<user.dial_code> ... </user.dial_code>
<user.phone> ... </user.phone>

<actor.actor_short_name> ... </actor.actor_short_name>
<actor.actor_legal_name> ... </actor.actor_legal_name>
<actor.country> ... </actor.country>
<actor.nif> ... </actor.nif>
<actor.business_simple_code> ... </actor.business_simple_code>
<actor.cep> ... </actor.cep>
<actor.email> ... </actor.email>

<item>
<item.lang> ... </item.lang>
<item.usage> ... </item.usage>
<item.category> ... </item.category>
</item>

</email_content>
```

De acordo com a especificação, o módulo deve assinalar erros quando os elementos estão mal estruturados, por exemplo, quando as *tags* não são fechadas/abertas. Por esta razão, o processador de linguagem tenta detetar erros de sintaxe de forma semelhante ao que um compilador de uma linguagem de programação faz.

Se não houvesse a necessidade de realizar o tratamento de erros, a implementação deste procedimento teria recorrido a expressões regulares para procurar e extrair a informação. A biblioteca *re* do *Python*, [14], permite efetuar operações sobre *strings* através de expressões regulares.

Atendendo à possibilidade dos dados de entrada, recebidos pelo módulo, não estarem bem formatados, o processador começa por converter o que recebe para um formato base. O processador de linguagem condensa primeiro o texto em uma *string* (sem perda da informação dos parágrafos), e normaliza as *tags* de abertura/fecho, isto é, retira espaços em volta dos nomes das *tags* e entre os caracteres '<' e '>' e converte os seus nomes em minúsculas. O exemplo 6 ilustra a remoção do espaçamento supérfluo das *tags* e a preservação dos parágrafos pela sequência de caracteres '\n'.

Exemplo 6: Singularização do texto e normalização do conteúdo

O seguinte texto de entrada contém *tags* com espaçamentos e caracteres maiúsculos.

```
[“<email_content>”,
“”,
“< EMail.Subject  > Assunto do email < / email.subject>”,
...
```

O primeiro passo do processador de linguagem normaliza as *tags* e converte o texto (lista de *strings*) em uma única *string*.

```
“<email_content>\n\n<email.subject> Assunto do email </email.subject>\n...”
```

No fim do passo anterior, o processador de linguagem extrai o conteúdo da *tag* <email_content>, descartando o resto do conteúdo presente no texto do *email*.

No passo seguinte, o processador de linguagem converte a *string* numa lista alternada de *tags* e texto (ver exemplo 7). Esta é feita através de um comando que separa *strings* com o critério de separação definido por uma expressão regular (ver listagem 8) que procura por padrões de *substrings* compostas por caracteres alfanuméricos (incluindo ponto final e o *underscore*), com pelo menos um carácter entre os caracteres '<' e '>' e, opcionalmente, uma barra '/' a seguir ao carácter '<'.

Se duas *tags* estiverem justapostas então aparece uma *string* vazia entre elas na lista.

Exemplo 7: Conversão da string numa lista de tags/conteúdo alternada

A seguinte *string* é o conteúdo da *tag* <email_contents> depois de ter sido singularizada.

```
“\n\n<email.subject> Assunto do email </email.subject>\n...”
```

Esta *string* é transformada em uma lista de *strings* com *tags* e o que existe no seu interior alternada:

```
[“\n\n”,
“<email.subject>”
“ Assunto do email “
“</email.subject>
“\n...”,
...
```

Listagem 8: Separação duma *string* numa lista alternada de *tags*

```
list_of_strings = re.split(r'(<[/]{0,1}[a-zA-Z0-9._]+>)', string)
```

Verificação da boa estrutura dos elementos

O processador de linguagem verifica se os elementos estão bem estruturados no conteúdo dos emails. Se as *tags* forem abertas/fechadas devidamente e não possuírem outras *tags* no seu interior que não lhe pertençam então estão bem estruturados.

Neste ponto do processamento, o conteúdo está arranjado numa lista alternada entre *tags* de abertura/fecho e texto, sendo fácil selecionar as *tags* para fazer a verificação.

A função de verificação da boa estrutura serve-se de uma pilha de *tags*:

- Quando lê uma *tag* de abertura, tenta adicioná-la à pilha. Se a pilha estiver vazia, ou se a *tag* a ser adicionada corresponde a um subelemento do elemento representado pela *tag* no topo da pilha então a *tag* é adicionada. Caso contrário, o programa assinala um erro indicando que este elemento não pode conter o subelemento.
- Quando lê uma *tag* de fecho, compara o seu nome com o nome da *tag* no topo da pilha. Se os nomes forem iguais então remove a *tag* que está no topo. Caso contrário, assinala um erro por tentar fechar uma *tag* com nomes de abertura e fecho diferentes.
- Quando atinge o fim da lista que analisa, a pilha pode não estar vazia. Se a pilha contiver alguma *tag* então é porque existe pelo menos uma *tag* que não foi devidamente fechada, assinalando também um erro.

Embora a maior parte dos elementos sejam de conteúdo simples, é possível que isto venha a mudar no futuro, portanto a função da verificação da boa estrutura funciona para o caso geral.

Rearranjando o conteúdo

Neste passo, o conteúdo do elemento <email_content> é rearranjado para uma melhor visualização da estrutura dos dados. De seguida, o conteúdo é separado em duas listas, uma com conteúdo válido e outra com conteúdo inválido. O conteúdo válido são elementos aceites pelo módulo. O conteúdo inválido é o restante, que inclui os elementos não aceites, bem como o texto entre os elementos.

Para rearranjar o conteúdo, a função concatena cada *tag* de abertura com o conteúdo do elemento e com a *tag* de fecho. Se um elemento tem conteúdo complexo, então as suas *tags* permanecem isoladas. O exemplo 8 ilustra este processo.

Os espaços imediatamente a seguir ao início e antes do final do conteúdo de cada elemento são removidos. À exceção de alguns elementos, como por exemplo as mensagens e as descrições de itens, os parágrafos são também removidos do conteúdo interior.

Exemplo 8: Rearranjando o conteúdo

A seguinte lista é separada por uma função em conteúdos válido e inválido.

Lista original:

```
[“<user.first_name>”,
“Rui“,
“</user.first_name>”,
“Isto é um comentário entre elementos\n”,
“<user.last_name>”,
“Bettencourt“,
“</user.last_name>”,
“\n”,
“<does_not_exist>”,
“Gambuzinos“,
“</does_not_exist>”,
“”,
“<item>”,
“”,
“<item.usage>”,
“BS”
“</item.usage>”,
“B: Buy, S: Sell”,
“</item>”,
...
```

Conteúdo válido:

```
[“<user.first_name>Rui</user.first_name>”,
“<user.last_name>Bettencourt</user.last_name>”,
“<item>”,
“<item.usage>BS</item.usage>”,
“</item>”,
...
```

Conteúdo inválido:

```
[“Isto é um comentário entre elementos”,
“<does_not_exist>Gambuzinos</does_not_exist>”,
“B: Buy, S: Sell”,
...
```

Preenchendo o dicionário de conteúdo

A fase de processamento de linguagem tem por finalidade devolver um dicionário com o conteúdo válido. Para este fim, existe uma função que converte os dados dentro do conteúdo (válido) para uma estrutura em dicionário.

Para cada elemento de conteúdo válido, o nome da *tag* é inserido no dicionário como chave e o seu conteúdo como valor, como demonstrado pelo exemplo 9. Quando a função adiciona um elemento ao dicionário, é testado se a sua chave já pertence ao dicionário. No caso afirmativo, o módulo assinala erro indicando que o elemento vem repetido nos dados de entrada.

Até ao momento de escrita deste relatório de estágio, existem apenas dois elementos de conteúdo complexo, sendo o <item> um exemplo. Adicionalmente, estes elementos podem aparecer repetidos nos dados de entrada. Como são casos particulares, estes elementos são tratados por funções dedicadas à extração do seu conteúdo.

Exemplo 9: Composição do dicionário

As *tags* na lista de conteúdos válidos são inseridas em um dicionário:

```
[<email.subject>Assunto do email</email.subject>,
...
```

```
contents['email_subject'] = "Assunto do email"
```

3.3 Análise de requisitos

A segunda fase do módulo executa uma análise de requisitos. A análise de requisitos começa por fazer a verificação da lista de operações que vêm definidas nos dados. As operações descrevem o que deve ser feito no resto do programa principal e estas têm interdependências entre si. Por exemplo, a criação de uma empresa requer que um utili-

zador seja criado, enquanto a criação de um catálogo requer que uma empresa seja criada.

Nesta fase, apenas é verificado o preenchimento dos campos obrigatórios relativos às operações validadas. Se a criação de uma empresa vem validada nas operações então os dados obrigatórios referentes à empresa devem estar todos preenchidos.

A função que realiza a verificação deve assinalar erros quando existe uma operação dependente de outra que está em falta, incluindo o que deve ser retificado na mensagem de erro. A função deve também assinalar erros quando existem campos obrigatórios em falta.

3.4 Validação dos campos

Na terceira fase, uma função valida os campos. A função de validação percorre o dicionário de conteúdos, apenas selecionando as chaves de operações validadas, realizando um controlo de qualidade sobre os dados.

A função desta fase contém algumas validações triviais, como por exemplo, verificar os tipos dos dados ou o comprimento do texto, no entanto as funções de validação mais complexas foram importadas a partir da biblioteca de *software* da centroProduto. As funções de validação incluem a validação de um NIF, do código duma língua, do *ID* das campanhas na base de dados, etc…

3.5 Composição da resposta

Quer os dados de entrada falhem em alguma das fases anteriores, quer sejam validados com sucesso, o programa devolve sempre um dicionário reestruturado como resposta. Este dicionário, de resposta, contém a mesma informação que o dicionário de conteúdos, mas subdividida por dicionários referentes às operações, aos erros e a dados extra (como por exemplo o resto do conteúdo do *email* original).

As *tags* dos elementos dos dados de entrada têm denominações que são dados por um módulo/programa que antecede o módulo de validação que é implementado neste capítu-

lo. No entanto, as chaves dos campos do dicionário de resposta devem coincidir com os nomes dos campos na base de dados para onde serão enviados.

Ao compor o dicionário de resposta deste módulo, as suas chaves passam por uma função de *aliasing* (de renomeação). A função de *aliasing* é ela também um dicionário que tem os campos de entrada como chave e os campos de saída como valores. A listagem 9 descreve a forma normal de implementar este tipo de funções, uma vez que, no *Python* não existe o *case of*.

Listagem 9: Dicionário da função de *aliasing*

A função de *aliasing* é composta por um dicionário que recebe um parâmetro de entrada como chave do dicionário e devolve o seu valor respetivo.

```
alias['user.email']      = 'email'
alias['user.first_name'] = 'first_name'
alias['user.last_name']  = 'last_name'
```

Se o valor de entrada for 'user.email', a função devolve (renomeia para) 'email'.

3.6 Testes

A funcionalidade *FASTINSERT()*, que inclui o módulo desenvolvido neste capítulo, foi testado por mim e pelos restantes membros da equipa. Quando um programa é composto por uma composição de módulos então os seus testes são feitos sobre o programa inteiro, ou em segmentos de módulos, por todos os membros.

Testámos a funcionalidade a partir do ponto onde a informação, em *XML*, é enviada via *email* para o módulo desenvolvido neste capítulo, até ao ponto em que o programa responde via *email* com a descrição do sucesso ou insucesso.

Para testar a funcionalidade *FASTINSERT()* foi-me concedido o acesso a um serviço de correspondência por *emails* por onde os dados de entrada eram enviados e por onde posteriormente recebia respostas. O envio dos dados no *email* obrigou a uma composi-

ção com a estrutura *XML* devida, bem como um assunto específico, o que fez com que os testes levassem bastante tempo na sua execução. Por outro lado, foi interessante experienciar o efeito que o módulo desenvolvido teve no portal.

Para agilizar os testes na versão local do módulo, desenvolvi uma interface gráfica (ver figuras 1 e 2) que lê um campo de texto, o processa e mostra o seu resultado noutro campo de texto. A versão local do módulo é a versão que não depende das funções da biblioteca de *software* da centroProduto. Este método de testes provou ser muito mais prático do que alterar ficheiros de texto para serem lidos como dados de entrada. Adicionalmente, este programa salienta as *tags* o que permite uma visualização mais prática da estrutura *XML* dos dados de entrada.

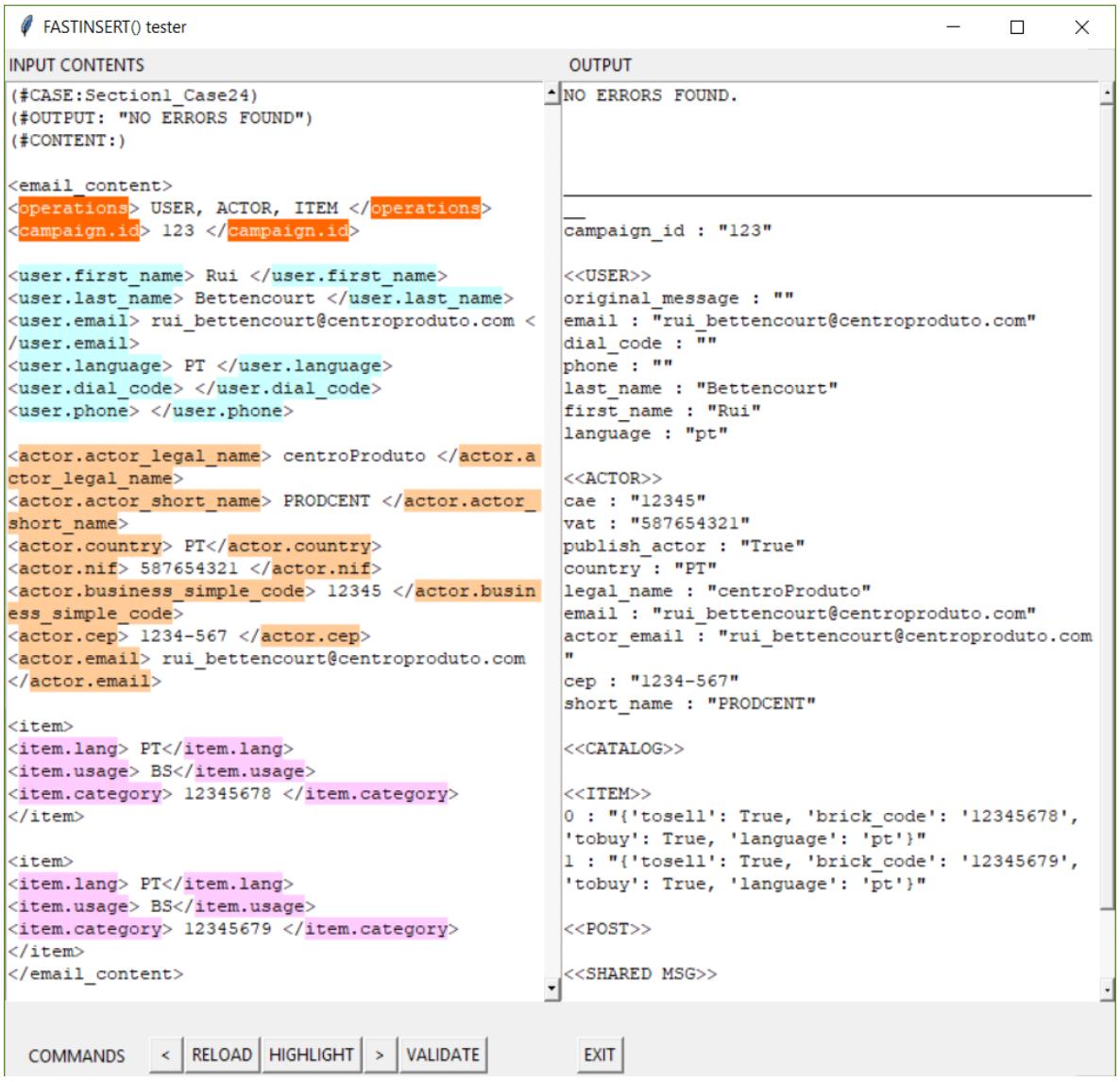

Figura 1: Interface do programa de testes locais devolvendo sucesso.

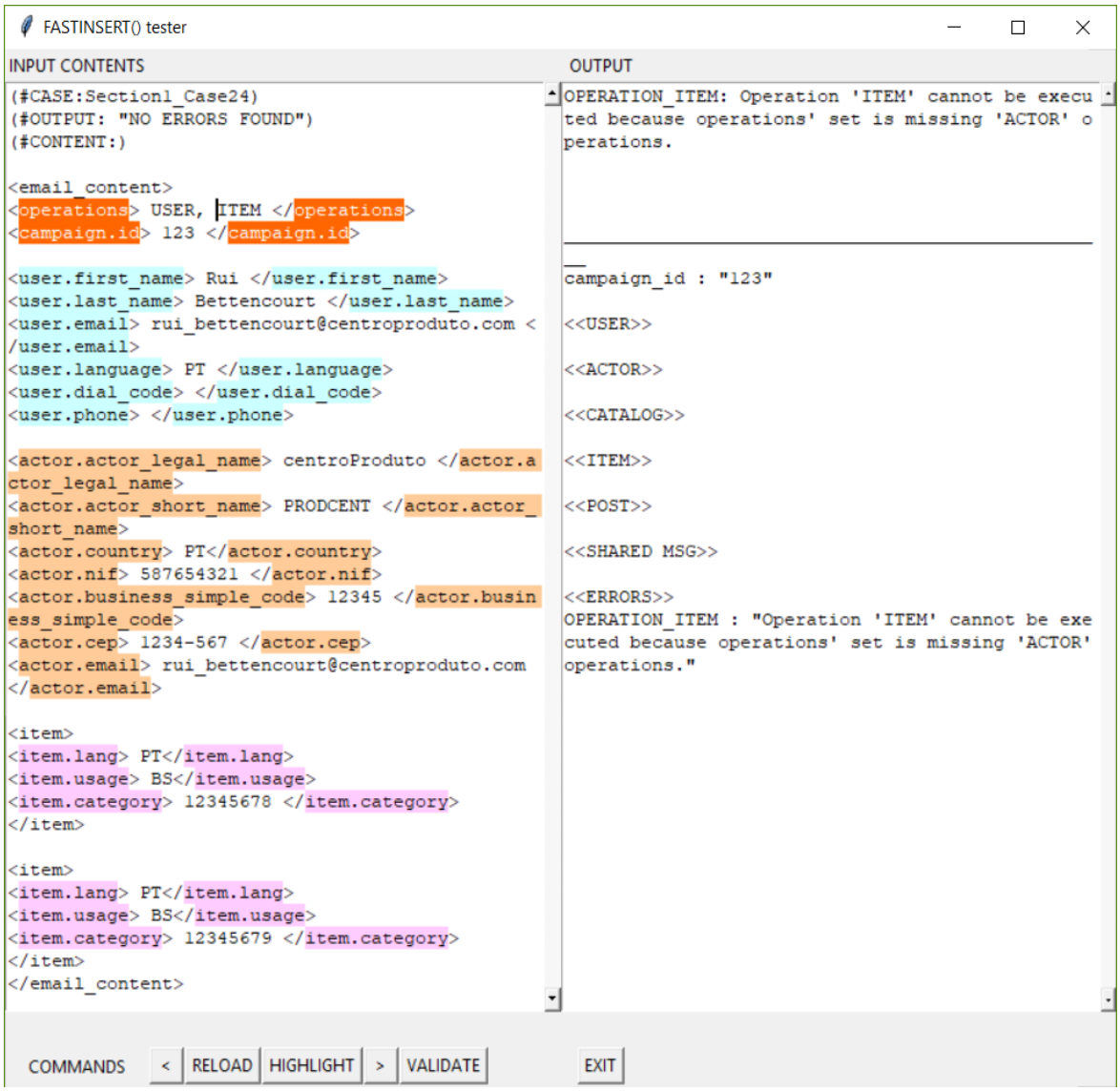

Figura 2: Interface do programa de testes locais devolvendo insucesso

4. Automatização de envio de mensagens

Objetivo: Automatizar o envio de mensagens para várias empresas no portal da centro-Produto.

Procedimento 3
1) Ler um ficheiro de entrada (que contém uma lista de *URL* relativas a empresas e uma lista mensagens em diversas línguas) construindo o suporte de dados. 2) Percorrendo a lista de *URL*, aceder à página-perfil das empresas e contactá-las.

4.1 Usabilidade e fluxo de operações do programa

A pedido do orientador de estágio, os programas que desenvolvo devem passar a ter uma interface mais prática para os utilizadores, tornando mais fácil a interação e a leitura da informação. As medidas que tomei para atingir este objetivo foram:

- Simplificar a interface e usar elementos visuais para transmitir melhor a informação.
- Reduzir a quantidade de informação apresentada ao utilizador e omitir a informação não relevante.
- Implementar mecanismos de carregamento de teclas em vez de submissões.

Usei caracteres de caixa ASCII (*ASCII box characters*) para compor formas com linhas verticais e horizontais contínuas na consola, o que deu um aspeto mais limpo à apresentação da informação. Para não cansar a vista ao utilizador, encurtei o texto das opções, listei-as verticalmente e dei-lhe um espaçamento de uma linha. A indicação do progresso do processo passou a ser feito graficamente, através de uma barra em vez de uma percentagem como o era originalmente.

A informação apresentada na consola foi condensada e a informação irrelevante, deixada como herança do desenvolvimento do programa, foi removida.

Através de uma biblioteca standard do *Python*, *msvcrt*, foi possível criar uma função que lê as teclas pressionadas na consola (detalhada na secção 7.3).

O menu inicial do programa, onde é escolhido o ficheiro de entrada e algumas configurações antes do arranque do envio de mensagens, foi a parte mais afetada por este tipo de melhoramentos. Para cada passo no menu inicial, é apresentada na consola uma breve e concisa mensagem, seguida de uma lista de opções com comandos associados a uma tecla.

Na escolha do ficheiro de entrada, o programa agora lista um menu com ficheiros (folha de cálculo) disponíveis no diretório onde o programa corre, conforme descrito na secção 7.3. Em vez de escrever o nome do ficheiro manualmente, como tem sido habitual nos programas criados até agora, o utilizador apenas tem de selecionar, com uma tecla, qual o ficheiro desejado. Este método evita que o utilizador perca tempo a escrever o nome do ficheiro por extenso.

No próximo passo do menu inicial, o utilizador escolhe qual a instância do portal que deseja utilizar. O objetivo do programa é enviar mensagens para clientes na instância real do portal, mas para conduzir testes e verificar que o programa age de acordo com o que é esperado, é oferecida a opção enviar pela instância de desenvolvedor.

No terceiro e último passo do menu inicial, o utilizador pode escolher o modo/tempo de espera do processo. O utilizador pode optar por escolher entre dois a nove segundos entre o envio de mensagens ou selecionar o modo de espera, que faz com que o programa espere por uma confirmação antes de executar o envio de uma mensagem.

O tempo de espera entre o envio de mensagens mínimo é de dois segundos para evitar um *bug* da página do portal. O modo de espera permite visualizar e verificar, sem tempo limite, o que vai no assunto e corpo da mensagem, antes de ser efetuado o seu envio. Este modo serve para verificar que o programa funciona bem na instância real, sem que haja consequências indesejadas.

4.2 Suporte de Dados

O programa lê uma folha de cálculo, como ficheiro de entrada, que contém duas folhas: uma com a lista de empresas a serem contactadas e outra com as mensagens em diversas línguas para serem usadas para contactar as empresas.

A lista de empresas tem associada a cada empresa (isto é, à *URL* da página perfil da empresa), um código de uma língua em formato de dois caracteres. Por exemplo, Portugal é descrito como PT. A lista de mensagens possui, para cada código de língua, o assunto e o corpo da mensagem. O exemplo 10 ilustra a estrutura da informação do ficheiro de entrada.

Exemplo 10: Estrutura da informação do ficheiro de entrada

A folha de cálculo (ficheiro de entrada) possui duas folhas: a *URL_list* e a *lang_list*, descrevendo seus os campos na coluna direita.

URL_list	
short_URL	*URL* reduzida da página perfil da empresa
language	Código da língua

lang_list	
Language	Código da língua
subject	Assunto
message	Mensagem

As folhas do ficheiro de entrada podem ser vistas como tabelas de uma base de dados. A língua é o campo comum que liga as tabelas, ou seja, a *language* na folha *URL_list* deve conseguir referenciar a *language* na folha *lang_list*. É através desta perspetiva que faço o tratamento dos dados e a verificação de erros.

O programa converte os códigos de línguas em ambas as folhas em minúsculas para evitar que erros de capitalização interfiram na referenciação.

Para verificar que as referências dos códigos de línguas existem, o programa começa por ler a folha que contém as mensagens por línguas (*lang_list*), ignorando linhas que não tenham assuntos ou corpos de mensagem. Só depois o programa lê a lista de empresas (*URL_list*), apenas admitindo aquelas que têm uma língua presente nas mensagens.

Ao ler as *URL* das empresas, o programa verifica se as mesmas têm formato de endereços e se são do portal centroProduto. Esta verificação não é crítica em termos de segurança, porém impede que o *webdriver*, aplicação que o programa usa para interagir com portal, perca tempo ao tentar (sem sucesso) aceder à *URL*. Se existir alguma empresa (*URL*) que não pode ser contactada, o programa regista o estado de insucesso do envio da mensagem incluindo a razão do mesmo.

4.3 Interagindo com o portal através do *webdriver*

Para interagir com o portal da centroProduto, o programa serve-se de uma biblioteca em *Python*, *Selenium*, introduzida no capítulo 2. Anteriormente, o *Selenium* e o webdriver foram utilizados pela necessidade de obter informação das páginas que apenas podia ser adquirida usando um *web browser*. Neste programa, o uso do *Selenium* deve-se à necessidade de interagir com os elementos das páginas do portal, como por exemplo, clicar em botões e escrever em caixas de texto.

Login

O programa começa por aceder à página inicial do portal da centroProduto e tenta fazer *login* como um utilizador. Neste passo, o programa tem de interagir com uma janela *pop-up* que pede a seleção do idioma.

Após o idioma selecionado, o programa procura a opção para fazer *login*. No *pop-up* de *login*, o programa preenche o *email* do utilizador automaticamente, o que facilita o desenvolvimento e melhora a usabilidade. De acordo com a especificação, o programa não deve conter (nem guardar) senhas para preencher automaticamente o seu campo, pelo que o utilizador as deve inserir manualmente.

Quando o programa abre o *pop-up* de *login* entra num ciclo de espera até que detete que o *webdriver* foi redirecionado para a página de "*Feed* de Negócios", que é a página para onde o portal redireciona um utilizador depois de fazer *login* com sucesso.

Iterando a lista de *URL*

Após fazer *login* com sucesso, o programa percorre a lista de *URL* das empresas a serem contactadas. As *URL* redirecionam o *webdriver* para as páginas de perfil das empresas. Na página perfil da empresa, o *webdriver* interage com o botão para aceder à página de contacto, simulando um clique do rato.

Na página de contacto, onde é possível compor o assunto e o corpo da mensagem, o programa espera dois segundos antes de inserir o assunto e mensagem. A espera de dois segundos serve para evitar que o preenchimento dos campos seja apagado enquanto a página não carrega por completo. O *Selenium* oferece mecanismos para este tipo de problemas, esperando que os elementos da página permitam a sua interação, no entanto não consegui arranjar uma solução mais elegante para este caso particular.

Dependendo da configuração do tempo de espera, o programa age de modo diferente. Se o programa está em modo de espera então antes de enviar a mensagem o utilizador tem de selecionar a opção continuar. Alternativamente, o utilizador pode escolher ignorar e saltar para a próxima empresa ou sair do programa. Se o programa está em modo de envio automático então as mensagens são enviadas em intervalos de alguns segundos. Quando o programa envia a mensagem, o programa regista o estado do envio da mensagem no ficheiro de saída.

4.4 Segurança e integridade do processo

O programa que desenvolvi interage diretamente com o portal da centroProduto, através de uma conta oficial, cujas mensagens são enviadas para clientes. Por este motivo, tentei evitar quaisquer erros que pudessem surgir e implementei alguns mecanismos de segurança.

Uma medida que implementei foi criar uma função que registasse, numa coluna denominada *delivery_status* (estado do envio) do ficheiro de saída, se a mensagem:

- Foi enviada com sucesso, registando como *SENT* (enviado).
- Foi ignorada, registando como *SKIPPED* (ignorado/saltado).
- Falhou a ser enviada, registando como *NOT SENT* (não enviado).

Adicionalmente, o programa regista a razão da falha junto do seu estado.

Quando o programa lê o ficheiro de entrada, avalia imediatamente se a empresa não pode ser contactada e regista o estado do envio nesse momento. O programa não deve contactar empresas com falhas nos dados de entrada, nem múltiplas vezes. Caso o programa falhe ao contactar uma empresa na lista, deve registar o estado do envio como insucesso, indicando que a deve voltar a tentar a contactar mais tarde.

Os ficheiros de saída podem ser reutilizados como ficheiros de entrada se o utilizador assim o desejar. O programa ao ler um ficheiro processado ignora as empresas cujo estado do envio foi assinalado como sucesso, prevenindo assim de a contactar múltiplas vezes. Quanto aos restantes estados, o programa reavalia se pode contactar as empresas. Desta maneira, o utilizador pode corrigir os erros no ficheiro para que o programa volte a tentar enviar as mensagens que previamente falharam.

As contas do portal centroProduto são compostas por um utilizador associado a uma ou mais empresas. Uma conta (ou utilizador) do portal que tenha múltiplas empresas pode selecionar em qual destas está correntemente ativo. O utilizador pode facilmente saltar entre empresas ativas, mas apenas uma pode estar correntemente selecionada.

A conta oficial do portal centroProduto, a que este programa se destina, não escapa a este molde. Por vezes, a conta oficial está a ser usada em múltiplas máquinas/*web browsers*/utilizadores. Se um utilizador mudar a empresa correntemente ativa, este efeito repercute-se nas outras máquinas. O programa supostamente deve enviar as mensagens pela conta oficial do portal centroProduto através de uma empresa ativa específica, por este motivo, tentei arranjar um mecanismo de segurança para este problema.

O portal centroProduto indica, através de um *banner* da empresa, qual a empresa ativa. Um *banner* de uma empresa é uma pequena imagem que a representa de forma semelhante a um logotipo. O *banner* da empresa contém uma *URL* que por sua vez contém o

seu *ID*. À data da escrita deste relatório de estágio, este *ID* é a única forma de identificar uma empresa através do código fonte das páginas no portal.

Pensei em implementar uma função que extrai a *URL* do *banner* e verifica se o *ID* pertence a uma empresa que deve estar ativa. A verificação deve ser feita mesmo antes de enviar a mensagem e logo após o envio, o que permite detetar a alteração da conta ativa no espaço de tempo que decorre entre o envio e o carregamento das páginas. Infelizmente, esta medida foi apenas implementada na instância de desenvolvedor. Na instância real não foi possível extrair o *ID* da empresa dentro do *banner* e não existem outras alternativas a não ser alterar a configuração do *website*, o que demoraria alguns meses a alterar e testar antes ser lançada.

4.5 Testes

Para agilizar os testes, criei um ficheiro de entrada com uma organização dos dados para testar os vários casos esperados, isto é, os casos em que o programa supostamente enviava as mensagens com sucesso e os casos em que falhava. Desta maneira, consegui agilizar o processo de testes para aprovar cada nova versão do programa em poucos minutos.

Tive o cuidado de inserir todos os casos que testam a funcionalidade do programa com os possíveis erros, dispostos de forma aleatória. Esta técnica permitiu testar as funcionalidades do programa, não descartando a possibilidade de falhas imprevisíveis que possam surgir.

4.6 Desempenho

Os resultados do desempenho obtidos pelo programa, de acordo com o próprio orientador de estágio da empresa foram os seguintes:

- Em 377 empresas, o programa enviou, com sucesso, para 362 empresas, falhou para 15 na primeira passagem.
- Na estimativa temporal sobre o tempo de conclusão de um processo de envio, o programa previu que o tempo seria de cerca de 1 hora e 57 minutos com um erro de 1 minuto apenas.

5. Base de dados

Neste capítulo começo por introduzir alguns conceitos de Bases de Dados, usando exemplos em *PostgreSQL*, [15], servindo-me da aplicação *pgAdmin4* e com excertos de código de programação em *Python*, fazendo uso da biblioteca *psycopg2*, [16], para interagir com a base de dados. Depois, apresento vários programas que desenvolvi que se centram na área de base de dados.

5.1 Algumas noções acerca de Bases de Dados

Uma base de dados, [17], é uma coleção organizada de dados que podem ser acedidos e manipulados. Os dados são organizados em tabelas. Nas tabelas, cada linha representa um registo e cada coluna (ou campo) representa um atributo.

O *PostgreSQL* é um sistema de gestão de bases de dados (SGBD) que usa a linguagem *SQL*. O *PostgreSQL* é um *software* livremente disponível para fins comerciais, sendo por esse motivo a escolha da empresa. Um SGBD é um tipo de *software* que permite definir, consultar e manipular dados e administrar uma base de dados.

O *pgAdmin4* é a aplicação com interface gráfica para interagir com a base de dados. Esta aplicação vem incluída na instalação do *PostgreSQL*.

O *psycopg2* é uma biblioteca em *Python* que fornece ferramentas para que um programa se possa conectar a uma base de dados e executar comandos. A vantagem de utilizar um programa para interagir com uma base de dados em comparação com um SGBD consiste na facilidade com que se pode estruturar e dinamizar os comandos de modo a tornar mais prático a manipulação dos dados.

Vistas

As vistas (*views*) são, de maneira muito simples, consultas que se podem chamar através de um nome. Quando se chama uma vista, esta executa a mesma instrução de consulta pela qual foi definida. Ao contrário das tabelas da base de dados, a informação das vistas

não existe fisicamente na base de dados. A sua informação é gerada no momento da sua chamada.

Uma das vantagens que as vistas têm sobre as consultas é que permitem um acesso aos dados mais controlado. Por exemplo, é possível através de uma vista aceder informação de maneira seletiva sobre uma ou várias tabelas.

Outra vantagem consiste na adaptabilidade das aplicações que interagem com a base de dados quando a sua configuração muda. Se a base de dados sofrer alterações ao seu esquema físico, todas as aplicações que executem consultas sobre a base de dados vão necessitar de ser atualizadas. No entanto, se as aplicações chamarem vistas e se a base de dados sofrer alterações então apenas será necessário atualizar as instruções nas vistas.

As vistas podem ser também chamadas a partir de outras vistas. Enquanto que este tipo de utilização traz vantagens quando usado com moderação, vistas chamadas a partir de outras vistas sucessivamente podem sobrecarregar o sistema de base de dados.

Triggers e funções

Os *triggers* são mecanismos desencadeados por eventos de manipulação de dados na base de dados que chamam funções. *Trigger* é a palavra Inglesa para gatilho, como o gatilho de uma arma, que quando pressionado, desencadeia uma reação que dispara uma bala. Na maior parte da sua aplicação, os *triggers* servem para fazer controlo de qualidade dos dados.

Existem restrições aos dados de uma BD que não podem ser cobertas pelos tipos de dados, feito na definição das tabelas. Por vezes, as restrições requerem que os dados obedeçam a condições ou sejam alterados através de funções, que podem ser chamadas pelos *triggers*.

Normalização

A normalização de uma base de dados é o processo de reestruturar o seu esquema físico de forma a mitigar a redundância dos dados, [18].

Uma base de dados não devidamente normalizada introduz problemas na manipulação dos dados, bem como aumenta desnecessariamente o espaço físico ocupado pelos dados na BD.

Exemplifico em seguida alguns problemas que podem ocorrer numa tabela não normalizada. A tabela 6 descreve alunos registados em disciplinas, com a informação de quem (um professor) é responsável pela disciplina.

aluno	**disciplina**	**professor_responsavel**
João Pedro	Base de Dados	Carlos José
João Pedro	Data Mining	Ana Rosa
Maria Isabel	Aprendizagem Computacional	Ricardo Daniel
Maria Isabel	Base de Dados	Carlos José

Tabela 6: Tabela não normalizada

A coluna *aluno* descreve o nome do aluno, a coluna *disciplina* descreve o nome da disciplina e a coluna *professor_responsavel* descreve o nome do professor que leciona a disciplina.

Nesta tabela os nomes dos alunos aparecem repetidos, conforme as disciplinas que frequentam. As disciplinas e os nomes dos professores responsáveis também aparecem repetidos.

Ao inserir vários alunos associados a uma mesma disciplina e professor, o nome da disciplina e do professor serão repetidos tantas vezes quanto o número de alunos inseridos, o que introduz dados redundantes. Para além disso, podem ocorrer gralhas na inserção dos nomes das disciplinas ou professores, o que causa inconsistência dos dados.

Se for necessário atualizar o nome do professor responsável de uma disciplina, será necessário atualizar múltiplos registos cuja coluna *professor_responsavel* é aquele que se pretende atualizar. Se for necessário remover um aluno ou disciplina, os dados relativos ao professor responsável podem perder-se.

Relações entre entidades

As tabelas-entidade podem estar associadas a outras tabelas-entidade. Estas associações chamam-se relações. Cada relação têm uma cardinalidade que indica quantos registos de uma tabela estão associados a quantos registos de outra tabela. Por exemplo, existem relações de:

- Um para um, onde um registo pode estar associado a apenas outro registo.
- Um para muitos, onde um registo pode estar associado a múltiplos registos.
- Muitos para muitos, onde vários registos podem estar associados a vários registos.

Estas relações são estabelecidas definindo colunas de referência adequadas. Os exemplos 11 e 12 ilustram a forma de definir relações de um para muitos e muitos para muitos, respetivamente.

Exemplo 11: Relação de um para muitos

As seguintes tabelas têm uma relação de um para muitos. Um classificador industrial tem vários códigos de atividade, mas cada código pertence a um só classificador.

tabela "classificador"
Id nome_do_classificador

tabela "código_de_atividade"
código_de_atividade classificador_id_ref (referência o atributo "id" da tabela "classificador")

Exemplo 12: Relação de muitos para muitos

As seguintes tabelas têm uma relação de muitos para muitos. Uma empresa tem vários códigos de atividade e cada código pode ter várias empresas.

tabela "empresa"
id
nome_da_empresa

tabela "código_de_atividade"
código_de_atividade
classificador_id_ref (referência o atributo "id" da tabela "classificador")

tabela "empresa__código_da_empresa"
empresa_id_ref (referência o atributo "id" da tabela "empresa")
código_id_ref (referência o atributo "id" da tabela "código_da_empresa")

Conexão e credenciais

Para que um utilizador se possa ligar a uma base de dados é necessário um conjunto de credenciais: o nome da base de dados, o nome do utilizador, a senha, o servidor e a porta de ligação.

O *pgAdmin4* pede estas credenciais quando um utilizador se tenta ligar a uma base de dados, sendo este processo feito apenas uma vez se se optar por guardar as credenciais localmente (na máquina).

Quanto aos programas que interagem com a BD, é necessário estabelecer uma ligação cada vez que o programa é executado. O *psycopg2* também fornece comandos próprios para este efeito. Transversal a todos os programas desenvolvidos neste estágio, criei um

pequeno módulo que lê as credenciais a partir de um ficheiro, à parte, para evitar que se exponham as credenciais dentro do código, bem como estar constantemente a introduzir as credenciais manualmente.

Atomicidade das operações

Existem situações em que é necessário executar uma sequência de comandos na base de dados de tal modo que, se um comando falhar então nenhum dos comandos restantes produz efeito na BD, ou seja, a sequência de instruções é executada de forma atómica.

Para exemplificar esta necessidade, falo sobre a inserção de um registo na base de dados de *marketing*/testes. Se quiser inserir um registo de uma empresa, na tabela *email*, e este tem associados outros dados em tabelas-entidade diferentes, como por exemplo as palavras-chave, tenho de executar três comandos:

- O primeiro comando insere dados na tabela *email*, e retorna o *ID* do novo registo recentemente criado.
- O segundo comando pode inserir uma palavra-chave, na tabela *keyword*, que ainda não exista na base de dados devolvendo *ID* do registo criado. Se a palavra-chave já existir então apenas devolve o *ID*.
- O terceiro comando usa os *ID* referentes às tabelas *email* e *keyword* para criar a relação entre o *email* e a palavra-chave.

Se o segundo comando, que introduz a palavra-passe, falha, não posso executar o próximo comando porque não tenho *ID* referente à palavra-chave. Também tenho interesse em que o registo referente à tabela *email* seja anulada porque está incompleto, e esse erro seja mais tarde anotado pelo programa que tenta inserir os dados.

Para conseguir concretizar as operações de forma atómica, os SGBD oferecem mecanismos próprios: comandos para fazer *commit* (efetuar as alterações) e *rollback* (anular as alterações).

Quando um comando de manipulação de dados é executado numa BD, este não produz efeito imediato, a não ser que o SGBD esteja configurado de forma diferente. Geralmente, os comandos de manipulação de dados apenas efetuam as alterações na base de dados quando o comando *commit* é chamado. O comando *rollback* desfaz as alterações

da base de dados que foram realizadas desde o último *commit* ou a partir do ponto que se estabeleceu a ligação.

Injeção de *SQL*

Para manipular dados numa base de dados de forma prática, é frequente usar aplicações que interagem com a base de dados. A maneira como as aplicações interagem com a base de dados consiste na geração dinâmica de comandos *SQL* para executar consultas ou inserir/atualizar/remover dados.

Existem duas abordagens para gerar os comandos dinâmicos de *SQL* na programação. Uma abordagem, talvez a mais intuitiva, consiste na concatenação de *strings* para criar comandos dinâmicos. Isto põe sérios riscos de segurança à base de dados por permitir que os comandos *SQL* sejam construídos (e executados) com código malicioso. Este código malicioso pode permitir que se aceda indevidamente aos dados da BD e até apagar/alterar os seus dados.

Outra abordagem consiste em passar os argumentos para os comandos *SQL*. Esta é a forma mais segura de interagir com a base de dados pois os argumentos são passados literalmente para os comandos, isto é, são passados e tratados como dados em vez de concatenados, e qualquer tipo de anomalia nos dados, como por exemplo código malicioso, não causa efeito na base de dados porque não é executado.

Exemplo 13: Injeção de *SQL*

A seguinte instrução retorna um registo da tabela *email* cujo *ID* é igual ao valor de 'user_input'.

```
cursor.execute("SELECT * FROM email WHERE id=" + str(user_input) + ";")
```

Se 'user_input' tomar um valor expectável de um *ID*, a instrução devolve um registo, mas se 'user_input' tomar o valor "1 OR 1=1" então retorna todos os valores da tabela *email*.

5.2 Bases de dados da centroProduto

Os programas desenvolvidos neste estágio operam sobre a base de dados de *marketing* da centroProduto, sendo desenvolvidos e testados na base de dados de testes.

A base de dados de *marketing* serve para guardar dados que auxiliam a operação de *marketing*. Esta base de dados é composta por cinco tabelas-entidade e quatro tabelas-relação. Entre as tabelas-entidade, a tabela *email* é a tabela *pivot* (principal) que se relaciona com todas as outras tabelas: *origin*, *keyword*, *responsibility* e *mobile*.

Todas estas tabelas têm relações de muitos para muitos (N:M) com a tabela *email*. Portanto, um *email* (representando, de maneira prática, uma empresa) pode ter associado várias palavras-chave (*keywords*), e uma palavra-chave pode estar associada a múltiplas empresas.

Esta base de dados, às vezes designada como *CP_MKT* (centroProduto *Marketing*), é aquela onde a informação recolhida pelo programa de *web scraping* é guardada.

A base de dados de testes possui o mesmo esquema físico que a base de dados de *marketing*, embora os dados sejam outros. Isto permite que os programas que são desenvolvidos e testados nesta base de dados funcionem garantidamente na base de dados de *marketing* sem qualquer problema.

A conta de utilizador que usei era bastante restrita nos privilégios e apenas me permitiu consultar e manipular dados na base de dados de teste. Para o desenvolvimento dos programas seguintes, tive de requisitar múltiplas vezes mais e mais privilégios para poder executar comandos necessários para testar as funcionalidades dos programas na base de dados de testes.

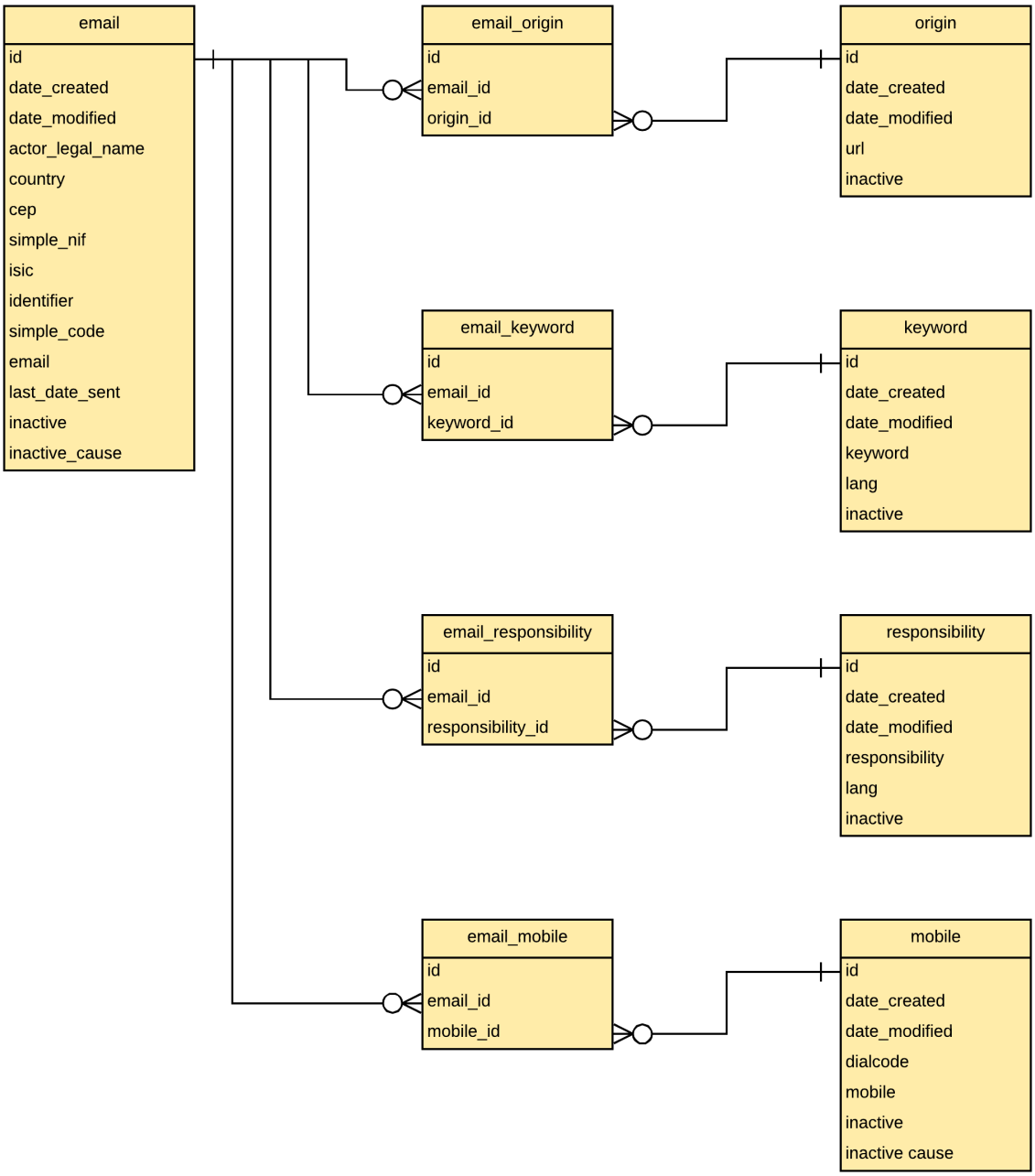

Figura 3: Esquema físico original da base de dados *CP_MKT*

5.3 Importação de dados para a base de dados de *marketing*

Objetivo: Inserir, ou atualizar, registos a partir de uma folha de cálculo para a base de dados de *marketing* da centroProduto.

Procedimento 4
1) Ler os registos do ficheiro entrada a partir de uma folha de cálculo. 2) Inserir, ou atualizar, os registos na base de dados, caso obedeçam aos requisitos necessários.

Nesta tarefa desenvolvi um programa em *Python* que insere registos na BD de *marketing* da centroProduto. Mais uma vez, esta tarefa também exigiu bastante atenção sobre como o programa afeta a base de dados. Não só a inserção de novos registos inadmissíveis podia causar alguma inconveniência pela necessidade da sua posterior remoção, como a atualização de registos já existentes podia danificar os dados para além da sua recuperação.

Processo de importação

O programa lê um ficheiro de entrada cujas linhas são associadas a endereços de *email*. O programa percorre o ficheiro, linha a linha, à medida que insere cada registo na base de dados. O ficheiro de entrada é uma folha de cálculo que tem colunas associadas a tabelas-entidade distintas na BD, ilustradas pela tabela 7.

Coluna da folha de cálculo	**Tabela da CP_MKT**
email_actor_legal_name	email
email_country	
email_cep	
email_simple_nif	
email_isic	
email_identifier	
email_simple_code	
email_email	

email_user_name	
origin_url	origin
keyword_keyword	keyword
keyword_lang	
responsibility_responsibility	responsibility
responsibility_lang	
mobile_dial_code	mobile
mobile_mobile	

Tabela 7: *Template* do ficheiro de entrada do programa de importação

Ao ler cada linha, o programa faz uma verificação sobre os campos obrigatórios e as suas dependências. Caso haja algum erro, o programa não insere o registo. O endereço de *email*, por exemplo, é um campo obrigatório. O número de telefone e o número prefixo do país, por exemplo, são interdependentes, ou seja, têm de estar ambos presentes ou ambos ausentes.

Se o endereço de *email* não existe na base de dados, o programa executa a sua inserção. Caso contrário, o programa tenta completar os dados em falta na BD com os dados presentes no ficheiro de entrada.

Atomicidade dos comandos de inserção

O programa original efetuava as alterações na base de dados após cada comando executado, usando uma opção chamada *auto-commit*, presente tanto no SGBD como nos programas através de uma opção que a biblioteca *psycopg2* oferece. Esta forma de introdução de dados permitia que, mesmo havendo falhas, os dados fossem efetivamente inseridos, mesmo que apenas parcialmente. Para evitar esta situação, o programa que desenvolvi regista as alterações quando completa a inserção dos dados de um registo em todas as tabelas.

Ordem de inserção

O programa opera primeiro sobre a tabela *pivot*, *email*, não só porque o processo de inserção/atualização depende fundamentalmente da existência do endereço de *email*, mas

também porque é necessário obter os *ID* respetivos para criar as associações entre as restantes tabelas-entidade.

Após operar sobre a tabela *email* e obter o respetivo *ID*, para cada tabela-entidade restante (*origin*, *keyword*, *responsibility* e *mobile*) o programa verifica se os dados já existem nas tabelas. Se já existirem, então apenas tem de obter o seu *ID*, se não existirem, então insere os dados e depois obtém o seu *ID*.

A combinação do *ID* associado ao registo da tabela *email* e o *ID* associado ao registo de outra tabela-entidade é o que permite criar a associação, nas tabelas associativas respetivas.

Registo do histórico

Por precaução, o programa regista todas as operações efetuadas na base de dados. Para este efeito, acrescenta a um ficheiro de texto mensagens com data e hora, a detalhar: o *status* da operação, a tabela afetada e o *ID* do registo inserido ou modificado. Isto é feito sempre que o programa executa uma instrução na base de dados. Este ficheiro serve como um histórico de operações efetuadas.

Garantia da preservação dos dados originais

De acordo com a especificação, o programa devia apenas inserir dados em campos da tabela *email* se estes estivessem vazios, sem alterar os campos preenchidos.

Para garantir esta condição, implementei uma função que, quando o programa atualiza a tabela, cria primeiro uma cópia integral do registo existente na base de dados e adiciona aos campos vazios dados provenientes do ficheiro de entrada se estes estiverem preenchidos. Durante este processo, o programa toma nota dos campos que foram alterados. Caso nenhum campo tenha sido alterado, o programa ignora e passa à frente. Caso existam novos dados, o programa altera o registo na BD sem qualquer alteração dos campos já preenchidos.

A atualização de um registo é feita à linha inteira da tabela, a partir de uma única instrução. Embora fosse possível alterar os dados de um registo campo a campo, tal método aumentaria o número de operações registadas no histórico.

Normalização dos dados

Não confundindo com a normalização das tabelas, a normalização dos dados refere-se ao tratamento que os dados sofrem antes de serem inseridos na base de dados.

Para que o utilizador, que executa a importação, não tenha a preocupação e necessidade de editar corretamente os valores a serem importados, o programa normaliza os dados automaticamente. Endereços de *emails*, códigos de línguas, *keywords*, *URL*, etc… são todos convertidos em minúsculas/maiúsculas, dependendo do caso.

Os campos relativos a outras tabelas-entidade que não a tabela *email*, como por exemplo as palavras-chave (*keyword*), são *strings*, que por vezes representam listas de palavras separadas por espaços ou vírgulas. Neste caso, o programa deve transformar este campo numa lista concreta, cujos elementos são também normalizados.

Testes

Para ajudar nos testes, criei um programa que gera um ficheiro de entrada com várias combinações de campos preenchidos. Por exemplo, uma linha tem todos os campos preenchidos para serem inseridos, outra linha tem todos os campos preenchidos menos o nome legal da empresa, outra linha não tem o *email* preenchido, e por aí fora…

Este método permitiu facilmente confirmar que o programa faz as verificações dos campos obrigatórios e das suas dependências corretamente. Adicionalmente, como o ficheiro contém centenas de registos, também foi possível observar como o programa interagia com a base de dados em inserções em massa.

5.4 Reorganização das palavras-chave

Nesta tarefa, criei um programa que reorganiza as palavras-chave (*keywords*) na base de dados.

Estado original das *keywords*

No estado original da base de dados de *marketing*, os registos das palavras-chave continham, no seu campo *keyword* múltiplas palavras-chave, separadas por vírgulas, pontos e vírgula, espaços, como uma *string* única. Este estado foi causado por uma utilização menos própria da base de dados, mais concretamente, a *string* do campo *keyword* era editada para incluir mais palavras-chave em vez de ser criado um registo novo.

Uma consequência desta forma de utilização de utilização é que se vários *emails* tivessem a mesma palavra-chave associada, a alteração da palavra-chave associada a um dos *emails* também se refletia nos outros *emails*.

Por exemplo, admita-se que existem dois *emails* para as empresas A e B, associados à mesma palavra-chave “carros, motas” (uma *string*). Se atualizar a palavra-chave para que a empresa A passe a incluir “camiões”, a empresa B também passa a estar associada a “camiões”.

A maneira correta de ter as palavras-chave, neste exemplo seria, as empresas A e B estarem associadas às palavras-chave “carros” e “motas” e criar uma associação entre a empresa A e a palavra-chave “camiões”.

De facto, a própria arquitetura da base de dados, pela tabela associativa *email_keyword* sugere que existe uma relação de muitos para muitos entre os *emails* e as palavras-chave, ou seja, um *email* podia estar associado a várias palavras-chave, uma palavra-chave podia estar associada a vários *emails*.

Consistência com a tarefa de importação de dados

Para que houvesse consistência na forma como os programas operam na base de dados, este programa e o anterior (de importação, da secção 5.3) tiveram de passar a fazer normalização dos dados de forma igual.

Backup

Este programa altera uma grande parte dos registos da tabela-entidade *keyword* e da tabela associativa *email_keyword*, que associa os *emails* às *keywords*.

Antes de arrancar com o processo, o programa deve guardar, em ficheiro, a informação das tabelas. Isto é feito antes de correr o processo de reorganização, com duas consultas que selecionam e guardam todo o conteúdo das duas tabelas. Por ser familiar e prático para os utilizadores na empresa, esta informação é também guardada em folha de cálculo, cada tabela numa folha própria.

Processo de reorganização

O processo de reorganização é feito em três momentos que dependem do tipo de operações a serem executadas. Estas operações, por sua vez, dependem do conteúdo da palavra-chave em cada registo da tabela-entidade *keyword*.

O programa começa por fazer uma consulta que seleciona todas a palavras-chave na base de dados, aplicando uma função de listagem a cada palavra-chave.

A função de listagem é uma função que recebe uma *string* como parâmetro de entrada e a separa por vírgulas ou espaços, devolvendo uma lista com palavras-chave singulares admissíveis. Uma palavra-chave é admissível se for composta por pelo menos três caracteres depois da normalização. A normalização de uma palavra-chave consiste em remover os seus caracteres especiais, a acentuação e em converter todos os caracteres em minúsculas.

A função de listagem pode devolver uma lista vazia se não existir uma palavra-chave admissível, uma lista singular (com um único elemento) ou uma lista com vários elementos. Isto influencia qual o tipo de operação que o programa realiza para cada registo original da palavra-chave na base de dados:

- Se a função devolve uma lista vazia, então a palavra-chave, e as respetivas relações, devem ser apagadas porque esta palavra-chave deixa de ser admissível depois da normalização.

- Se a função devolve apenas um elemento que após ser normalizado é diferente do registo (palavra-chave) original então o programa simplesmente tenta fazer uma atualização à palavra-chave na base de dados, evitando ter de operar sobre a tabela de associações. No entanto, se o elemento depois de normalizado coincidir com uma palavra-chave já existente, o programa deve fundir as associações de ambas.
- Se a função devolve múltiplos elementos então, caso não existam, são inseridos na BD. Para cada novo registo, são criadas as associações (*email* - palavra-chave) e o registo da palavra-chave original é posteriormente apagado.

Exemplo 14: Reorganização das palavras-chave

Admita-se que estes são os registos de palavras-chave na tabela *keyword*:

id	**keyword**
10	“V8”
20	“Carros”
21	“carros”
22	“Motas”
30	“Aviões, barcos, carros”

Ao passar a função de listagem pelas palavras-chave da tabela *keyword*, obtêm-se os seguintes resultados:

id	**keyword**	**listagem**	**número de elementos**
10	“V8”	[]	0
20	“Carros”	[“carros”]	1
21	“carros”	[“carros”]	1
22	“Motas”	[“motas”]	1
30	“Aviões, barcos, carros”	[“avioes”, “barcos”, ”carros”]	3

No primeiro momento, o programa trata os registos que devolvem 0 elementos na função de listagem:

Id	keyword	listagem	número de elementos
10	“V8”	[]	0

“V8” tem apenas dois caracteres, por isso não é uma palavra-chave admissível. A função de listagem devolve, portanto, uma lista vazia. Este registo deverá ser apagado como também as associações entre *emails* e a palavra-chave.

No segundo momento, o programa trata dos registos que devolvem apenas um elemento na função de listagem:

Id	keyword	listagem	número de elementos
20	“Carros”	[“carros”]	1
21	“carros”	[“carros”]	1
22	“Motas”	[“motas”]	1

“Carros” (id=20) ao ser normalizado fica como “carros” (com o ‘C’ minúsculo), mas “carros” (id=21) já existe na base de dados.

Portanto, o programa seleciona os *ID* dos *emails* associados à palavra-chave “Carros” (id=20) e cria as associações, entre estes *emails* e a palavra-chave já existente, “carros” (id=21). De seguida apaga a palavra-chave “Carros” (id=20) e as suas associações.

“carros” (id=21) ao ser normalizado fica igual, por isso não é necessário fazer nada. O programa passa para a próxima *keyword*.

“Motas” (id=22) ao ser normalizada fica como “motas”, como não existe nenhum conflito com outra palavra-chave existente na base de dados, o programa simplesmente atualiza o campo da palavra-chave (*keyword*) na base de dados, sem necessitar executar mais operações.

No terceiro momento, o programa trata dos registos que devolvem mais do que um elemento na função de listagem:

id	keyword	listagem	número de elementos
30	“Aviões, barcos, carros”	[“avioes”, “barcos”, ”carros”]	3

A palavra-chave “Aviões, barcos, carros” devolve uma lista com três elementos. Admita-se que existem associados a esta palavra-chave os seguintes (*ID* dos) *emails*:

email_id	keyword_id
1000	30
1001	30
1002	30

Estes são os *ID* dos *emails* (empresas) associados à palavra-chave “Aviões, barcos, carros” (id=30).

O programa guarda os *ID* destes *emails* e introduz as palavras-chave resultantes da função de listagem, também guardando os seus *ID* respetivos.

Id	*keyword*
40	“avioes”
41	“barcos”
21	“carros”

A palavra-chave “carros” já existia, com *ID* 21, “avioes” e “barcos” foram recentemente inseridos com os novos *ID* auto incrementais 40 e 41.

De seguida, o programa deve criar as várias combinações de associações entre os *ID* dos *emails* associados à palavra-chave original e as palavras-chave resultantes da função de listagem:

	40 (“avioes”)	41 (“barcos”)	21 (“carros”)
1000	(1000, 40)	(1000, 41)	(1000, 21)
1001	(1001, 40)	(1001, 41)	(1001, 21)
1002	(1002, 40)	(1002, 41)	(1002, 21)

Cada par nesta tabela é uma associação entre o *ID* referente ao *email* e o *ID* referente à nova palavra-chave, que deverá ser inserido na tabela associativa *email_keyword*. Feito isto, ainda é necessário apagar a palavra-chave original “Aviões, barcos, carros”, com *ID* 30, mais as respetivas associações.

5.5 *Backup* da base de dados

Objetivo: Extrair o conteúdo de uma base de dados para um ficheiro folha de cálculo.

Procedimento 5
1) Obter os nomes das tabelas na base de dados. 2) Para cada tabela, extrair o seu conteúdo para uma folha.

Foram-me atribuídas tarefas que envolviam realizar alterações na BD não só sobre os dados como também sobre a arquitetura. Para assegurar a preservação dos dados, a pedido do orientador da empresa, construí um programa que faz uma extração completa dos dados de uma BD.

Existem maneiras de fazer *backup* das bases de dados inerentes aos SGBD que vão mais além do que fazer uma simples cópia dos seus dados. O programa que desenvolvi extrai apenas o conteúdo das tabelas e os nomes das colunas, mas não guarda a definição das tabelas, das funções, dos *triggers*, dos índices, dos comandos executados, etc.

Este programa guarda todos os dados de uma BD de maneira mais acessível e visível aos utilizadores da empresa, tendo também as seguintes vantagens:

- É mais simples correr do que as funcionalidades existentes no SGBD.
- Extrai rapidamente todos os dados da base de dados, podendo ser usado para fazer *backups* periódicos.
- Funciona para qualquer BD, não dependendo da sua arquitetura.

O programa começa por executar o comando na listagem 10, que devolve a lista com o nome das tabelas da BD a que está ligado.

Listagem 10: Comando *SQL* para obtenção da lista de tabelas

```
SELECT tablename FROM pg_catalog.pg_tables
WHERE schemaname = 'public' ORDER BY tablename ASC;
```

De seguida, percorre a lista dos nomes das tabelas, executando a instrução da listagem 11, onde *tablename* é o nome de uma tabela.

Listagem 11: Comando *SQL* para obtenção dos registos de uma tabela

```
SELECT * FROM tablename;
```

Os dados das tabelas, incluindo os nomes das colunas, são guardados em folhas individuais do ficheiro de *backup* com os mesmos nomes das tabelas.

Nos comandos *SQL* dinâmicos usados até agora as variáveis foram sempre passadas literalmente como parâmetros, nunca sendo construídos via concatenação de *strings*. Conforme explicado anteriormente, a utilização de parâmetros evita ataques de injeção de *SQL*. No entanto, a única maneira possível para iterar sobre a lista de tabelas é via construção de *strings* concatenadas. O *tablename* no comando da listagem 11 não é um valor possível de ser passado como parâmetro. Esta é única exceção à maneira como construo as instruções *SQL* em todos os programas deste estágio.

Uma vez que os nomes das tabelas usados na listagem 10 são obtidos pelo programa a partir da BD e como as tabelas não podem possuir nomes que correspondam a instruções que afetem a BD então, em princípio, não há problema em utilizar instruções dinâmicas via concatenação.

Por precaução, como o código pode ser reaproveitado por outros colegas, adicionei uma verificação de que a tabela existe na BD, ver listagem 12.

Listagem 12: Mecanismo de segurança (*Python*)

```
cursor.execute("""SELECT 1 FROM information_schema.tables WHERE table_schema =
'public' and table_name=%s LIMIT 1""", [table_name])

results = cursor.fetchall()
if len(results) < 1: exit()
```

5.6 Associação de múltiplas atividades empresariais

Objetivo: Alterar a arquitetura da base de dados de forma a que seja possível associar múltiplas atividades empresariais a uma empresa (*email*).

Na BD de *marketing* da empresa, os dados acerca da atividade empresarial de uma empresa são guardados em algumas colunas na tabela *email*. Estas colunas são:

- O *isic*, que representa um código do standard de classificação industrial internacional (ISIC).
- O *simple_code*, que é um código de atividade associado a um standard (coluna identifier).
- O *identifier*, que representa o nome de um standard de classificação industrial.

Nesta configuração, um *email* pode ter associado um ISIC e um outro código de atividade (associado a um standard de classificação) qualquer, mas apenas um. Por outro lado, o ISIC também é um standard.

Esta estrutura não é a maneira ideal para guardar as atividades empresariais porque restringe o leque de classificações de uma empresa a um código apenas. Para além disto, não existe dependência entre o standard e o código de atividade.

O ideal seria criar uma tabela associativa que permita uma relação de muitos para muitos entre *emails* e códigos de atividades.

Nesta tarefa alterei a arquitetura da BD e transferi os dados referentes aos standards/atividades na tabela *email* para as novas tabelas criadas.

Para o processo de transferência de dados, não houve a necessidade de recorrer a programas pois era possível fazê-lo diretamente com instruções *SQL* pela aplicação *pgAdmin4*.

Alteração da arquitetura da base de dados de *marketing*

Para que um *email* possa estar associado a múltiplas atividades, criei uma tabela activity, que lista as atividades e uma tabela *email_activity*, que faz as associações entre *emails* e

atividades. Para agrupar as atividades por standard de classificação, também criei uma tabela, *industrystandard*, para ser referenciada pela tabela *activity*. A figura 4 descreve o modelo físico da nova arquitetura da base de dados.

Os nomes para as tabelas e para as colunas foram atribuídos de acordo com o padrão da arquitetura original com seguindo as boas práticas de bases de dados. Como existem códigos de atividade iguais pertencentes a standards de classificação diferentes, a chave primária da tabela dos códigos de atividade deve ser uma composição entre o código de atividade e o *ID* do standard de classificação.

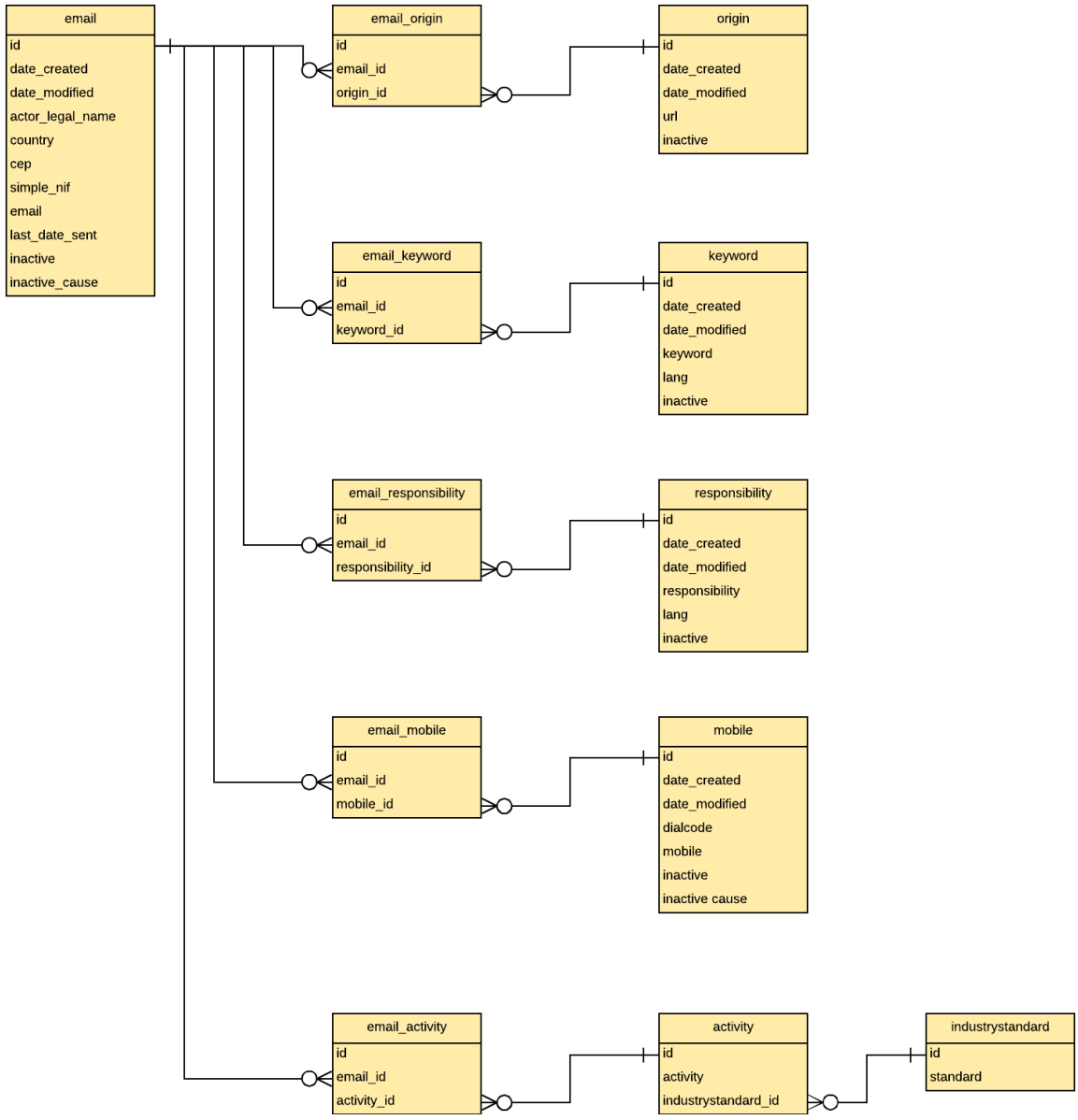

Figura 4: Esquema físico da BD de *marketing* alterada

Transferência dos dados para as novas tabelas

A transferência foi feita em dois tempos: um para a coluna de atividades do ISIC, ver listagem 13, e outro para as atividades dos restantes classificadores, ver listagem 14.

Para inserir registos na nova arquitetura, comecei pelas tabelas que não dependiam de referências a registos de outras tabelas. Não seria possível inserir dados na tabela *email_activity*, porque ainda não existem as referências para as atividades na tabela *activity*. Similarmente, não seria possível inserir dados na tabela *activity* porque ainda não existem referências para os seus classificadores de atividade na tabela *industrystandard*.

Listagem 13: *SQL* para a importação das tabelas (ISIC)

```
/* FILL TABLES (ISIC)*/
INSERT INTO email_activity (email_id, activity_id)
SELECT email.id, activity.id
FROM email
INNER JOIN activity ON (email.isic = activity.activity)
INNER JOIN industrystandard ON (activity.industrystandard_id = industrystand-
ard.id)
WHERE UPPER(industrystandard.standard) = 'ISIC_REV_4'
ORDER BY email.id;
```

Listagem 14: *SQL* para a importação das tabelas (restantes classificadores)

```
/* FILL TABLES (REST) */
INSERT INTO email_activity (email_id, activity_id)
SELECT email.id, activity.id
FROM email
INNER JOIN activity ON (UPPER(email.simple_code) = UPPER(activity.activity))
INNER JOIN industrystandard ON (activity.industrystandard_id = industrystand-
ard.id)
WHERE UPPER(email.identifier) = UPPER(industrystandard.standard)
ORDER BY email.id;
```

Sugeri ao orientador de estágio que os dados dos classificadores e dos respetivos códigos de atividade fossem pré-carregados para a BD em vez de serem inseridos cada vez que aparecem como novidade.

Ao inserir (ou atualizar) a um registo de uma empresa em que atividade é afetada, a integridade dos dados é respeitada se o registo for associado a um par código/classificador existente na BD. Assim, o controlo de qualidade dos dados é feito ao nível da BD.

O método alternativo seria inserir um novo código e/ou classificador na BD quando estes aparecessem como novidade na inserção (ou atualização) dum registo, no entanto, seria necessário fazer o controlo de qualidade externamente à BD.

Para realizar testes sobre a estrutura dos dados na tabela e verificar a transferência correta dos dados, criei à parte um pequeno programa que extrai os códigos (e descrições) das atividades, para cada classificador, a partir de compêndios disponíveis na *internet*.

Este programa constrói um ficheiro que lista os códigos de atividade por folha. Posteriormente fui informado que já exista informação sobre os vários classificadores, já tratada, o que acelerou o processo.

Desenvolvi também outro pequeno programa que trata de inserir os standards/atividades a partir de um ficheiro para as novas tabelas de atividade e classificadores na BD.

Após ter os standards e os seus códigos de atividade carregados, criei as associações entre os *emails* e as atividades, tendo a certeza que as atividades coincidiam com o classificador certo, caso contrário arriscava criar múltiplas associações a atividades com códigos iguais pertencentes a classificadores distintos.

Verificação da organização correta dos dados

Foi necessário confirmar que os dados transferidos foram devidamente associados nas novas tabelas. Antes de apagar as colunas isic, *simple_code* e *identifier*, da tabela *email*, onde os dados originalmente estavam, construí consultas que continham os dados antigos e os dados importados das novas tabelas para os comparar com os registos de atividades associados.

As listagens 15 e 16 fazem a verificação para o ISIC e restantes classificadores, respetivamente. A listagem 17 devolve registos que têm um classificador/código na coluna origi-

nal, mas que não foi associado na nova tabela. Os registos que caem neste resultado ou não existem ou então significa que o seu classificador não foi carregado na BD.

Como no estado original da BD cada *email* tinha no máximo um ISIC e uma atividade/standard, foi mais simples verificar que cada registo estava corretamente transferido, comparando os campos linha a linha.

Listagem 15: *SQL* para a verificação dos dados (ISIC)

```
/* CHECK TABLES */
--ISIC
SELECT email.email, industrystandard.standard, email.isic AS email_isic, ac-
tivity.activity
FROM email
LEFT OUTER JOIN email_activity ON (email.id = email_activity.email_id)
LEFT OUTER JOIN activity ON (email_activity.activity_id = activity.id)
LEFT OUTER JOIN industrystandard ON (activity.industrystandard_id = indus-
trystandard.id)
WHERE industrystandard.standard = 'ISIC_REV_4';
```

Listagem 16: SQL para a verificação dos dados (outros classificadores)

```
--REST
SELECT email.email, email.identifier AS email_identifier, industrystand-
ard.standard, email.simple_code AS email_simple_code, activity.activity
FROM email
LEFT OUTER JOIN email_activity ON (email.id = email_activity.email_id)
LEFT OUTER JOIN activity ON (email_activity.activity_id = activity.id)
LEFT OUTER JOIN industrystandard ON (activity.industrystandard_id = indus-
trystandard.id)
WHERE industrystandard.standard != 'ISIC_REV_4' OR industrystandard.standard
IS NULL;
```

Listagem 17: SQL para a verificação dos dados (Atividades não ligadas)

```
--MISMATCHED
SELECT email.email, email.identifier AS email_identifier, industrystand-
ard.standard, email.simple_code AS email_simple_code, activity.activity
FROM email
LEFT OUTER JOIN email_activity ON (email.id = email_activity.email_id)
LEFT OUTER JOIN activity ON (email_activity.activity_id = activity.id)
LEFT OUTER JOIN industrystandard ON (activity.industrystandard_id = indus-
trystandard.id)
WHERE email.identifier IS NOT NULL AND industrystandard.standard IS NULL;
```

5.7: Outras correções na base de dados

Correção dos registos de repetidos

Os programas de importação de dados para a base de dados de *marketing* que a empresa originalmente usava inseriram endereços de *emails* repetidos. Embora o campo do endereço de *email* devesse ser um registo único, a maneira como este estava definido na BD permitia que *emails* que variavam apenas na capitalização dos caracteres coexistissem na tabela. A BD estava desprovida de mecanismos que impedissem que esta situação acontecesse. Os programas de importação originais não normalizavam nem testavam a existência dos endereços de *email* que tentavam inserir na BD.

O programa de importação que desenvolvi no capítulo 5.3 evita inserir registos repetidos. Antes do programa inserir um registo, como por exemplo, um endereço de *email*, procura primeiro na BD por um registo (supostamente) único, que não é um *ID*. Para fazer esta verificação, o programa executa uma consulta convertendo os campos em minúsculas antes de os comparar. O programa atualiza as tabelas apenas quando estas retornam apenas um resultado.

Por exemplo, se ao procurar pelo endereço de *email* "rui_bettencourt@centroproduto.com", existissem na BD os *emails* "rui_bettencourt@centroproduto.com" e "Rui_Bettencourt@centroproduto.com", o programa assinala um erro pois a consulta retorna mais do que um resultado.

Embora este programa evite a inserção de novos registos repetidos na BD nem propague o problema, foi necessário corrigir este problema na BD.

No que se segue, demonstro como resolvi o problema de *emails* repetidos na BD de *marketing*, recorrendo apenas a algumas instruções *SQL* pelo *pgAdmin4* para apagar os registos repetidos e posteriormente usando o programa de importação do capítulo 5.3 para reinserir os registos apagados.

Os motivos pelos quais evitei usar a programação foram a rapidez com que pude resolver o problema usando apenas instruções *SQL* e porque poupei tempo no desenvolvimento de um novo programa e nos seus testes.

O plano de operações para corrigir a base de dados foi:

1) Obter e guardar os registos com *emails* repetidos.
2) Apagar os registos com *emails* repetidos da base de dados.
3) Reinserir os registos pelo programa de importação.

Criei uma vista, exemplificada na listagem 18, que imita a disposição dos dados no *template* do ficheiro de entrada do programa de importação.

Listagem 18: Criação da vista que imita o *template*

```
CREATE VIEW extended_xlsx_original AS
SELECT email.id, email.date_created, email.date_modified,
       email.actor_legal_name, email.country, email.cep,
       email.simple_nif, email.isic, email.identifier,
       email.simple_code,
       email.email, email.user_name, email.last_date_sent,
       email.inactive, email.inactive_cause,
       array_agg(DISTINCT origin.url) AS origins,
       array_agg(DISTINCT keyword.keyword) AS keywords,
       array_agg(DISTINCT responsibility.responsibility) AS responsibilities,
       array_agg(DISTINCT mobile.mobile) AS mobiles
FROM email
LEFT JOIN email_origin ON email.id = email_origin.email_id
LEFT JOIN origin ON email_origin.origin_id = origin.id
LEFT JOIN email_keyword ON email.id = email_keyword.email_id
LEFT JOIN keyword ON email_keyword.keyword_id = keyword.id
LEFT JOIN email_mobile ON email.id = email_mobile.email_id
LEFT JOIN mobile ON email_mobile.mobile_id = mobile.id
LEFT JOIN email_responsibility ON email.id =
          email_responsibility.email_id
LEFT JOIN responsibility ON
          email_responsibility.responsibility_id =
          responsibility.id
GROUP BY email.email;
```

Esta vista contém os dados de todas as colunas da tabela *email*, bem como os dados pertencentes às restantes tabelas associadas, dispostos em lista. Por exemplo, se um *email* estiver associado a duas palavras-chave, essas palavras-chave aparecem em lista numa coluna chamada *keywords*. O mesmo se aplica para os dados nas tabelas *origin*, *mobile* e *responsibility*.

Listagem 19: Consulta que devolve todos os registos com *emails* repetidos

```
SELECT * FROM extended_xlsx_original WHERE lower(email) IN (
    SELECT lower(email)
    FROM email
    GROUP BY lower(email)
    HAVING count(*) > 1
);
```

Selecionei os registos com *emails* repetidos para a vista que criei, ver listagem 19. A consulta interior (indentada) lista todos os endereços de *email* convertidos em minúsculas, depois agrupa por endereço de *email* e usa função de agregação de contagem para filtrar os *emails* idênticos agrupados com mais do que uma ocorrência. Esta consulta retorna uma lista de *emails* distintos que aparecem repetidos na base de dados, que diferem entre si na capitalização.

A consulta exterior seleciona os registos cujos endereços de *email* (em minúsculas) estão contidos na consulta interior. Assim, obtenho a lista de endereços de *email* repetidos no seu estado original, com todos os seus dados associados.

Para apagar os dados repetidos bastou usar o mesmo critério da consulta anterior selecionando os registos a serem removidos pelos seus *ID*, ver listagem 20.

Listagem 20: Instrução para apagar registos com *emails* repetidos

```
DELETE FROM email WHERE email.id IN(
    SELECT id FROM extended_xlsx_original WHERE lower(email) IN (
        SELECT lower(email)
        FROM email
        GROUP BY lower(email)
        HAVING count(*) > 1
    )
) RETURNING *;
```

Finalmente, usei o programa de importação, descrito na secção 5.3, para voltar a inserir os dados relativos aos *emails* repetidos. O programa de importação não deixa que *emails* repetidos sejam inseridos, em vez disso, apenas junta a informação em falta aos *emails* que já existem na BD.

Controlo de qualidade e tratamento dos dados

Os programas desenvolvidos neste capítulo, introduzem os registos tratados como é desejável. Por outro o ideal seria que a BD possuísse mecanismos de controlo de qualidade e tratamento dos dados.

Se controlo de qualidade e tratamento dos dados for feito na BD então não é necessário realizá-los sempre nas aplicações que interagem com a BD. Alternativamente, se as aplicações ficassem incumbidas de realizar estas tarefas então seria necessário que todas elas afetassem os dados de forma igual.

Nesta secção criei funções e *triggers* que fazem controlo de qualidade e tratamento dos dados na BD de *marketing*.

Anteriormente, os endereços de *email* na BD foram todos convertidos em minúsculas. Para evitar que endereços de *email* repetidos sejam inseridos na BD, criei uma função que converte um endereço de *email* em minúsculas. A seguir crio um *trigger* que chama a função anterior quando um registo é inserido ou atualizado na tabela *email*. Ver listagem 21. O *trigger* é desencadeado antes da inserção/atualização executando a função sobre o endereço de *email*, só depois de convertido em minúsculas tenta inserir/atualizar a tabela.

Listagem 21: Criação da função e *trigger* para controlo de qualidade dos dados

```
--TRIGGER FUNCTION
CREATE OR REPLACE FUNCTION f_convert_to_lower_email() RETURNS TRIGGER AS
$$ BEGIN
        NEW.email = LOWER(NEW.email);
        RETURN NEW;
    END; $$
LANGUAGE plpgsql;

--TRIGGER
CREATE TRIGGER t_convert_to_lower_email BEFORE INSERT OR UPDATE ON email FOR
EACH ROW EXECUTE PROCEDURE f_convert_to_lower_email();
```

Outros *triggers* semelhantes ao exemplificado na listagem 19 foram criados sobre os campos das *URL*, dos códigos de língua e palavras-chave para os converter em minúsculas.

6. Regularização da base de dados

Objetivo: Desativar *emails* em desuso e inserir *emails* de potenciais utilizadores atraídos pela campanha de *marketing*.

Neste capítulo desenvolvi um programa que regulariza a BD de *marketing*. O programa lê a partir de um ficheiro que contém respostas de *email*, em consequência da campanha de *marketing*, que devem ser usados para desativar endereços de *email* ou adicionar novos endereços de *email* para fins publicitários.

Antes de iniciar esta tarefa, já tinha trabalhado numa tarefa de classificação de *emails* de resposta. Estes *emails* eram respostas a campanhas de *marketing*. Foi desenvolvido um programa que lê um ficheiro extenso com muitos *emails* e os tenta classificar. Grande parte desta classificação foi realizada através da identificação de códigos de erro, bem como alguns outros indicadores únicos presentes no *email*. Os códigos de erro dos *emails* referem-se a um código composto por três dígitos, separados por dois pontos, que os servidores por onde os *emails* circulam inserem quando não têm sucesso. Neste contexto, um identificador único é um pedaço de informação usado para classificar inequivocamente um *email*. Por exemplo, uma mensagem pode indicar que uma resposta foi enviada automaticamente numa linha de informação inserida pelo servidor.

A partir dos códigos de erro e dos identificadores únicos foi possível classificar os *emails* que continham erros, os que eram autorrespostas, os que eram classificados como *SPAM*, e os que tinham sido enviados para endereços não existentes.

A tarefa de classificação de *emails* tinha dois objetivos:

- Adquirir novos endereços de *email*, contidos nas respostas, para serem sujeitos a publicidade pela centroProduto.
- Desativar endereços de *email* na BD que entraram em desuso, para evitar obter mais respostas de insucesso.

O objetivo do programa que desenvolvi neste capítulo é desativar e adicionar novos endereços de *email* na BD de *marketing* da empresa a partir do ficheiro de *emails* classificados.

De acordo com a especificação, o utilizador que opera este programa deve operar uma interface que lhe permita tomar decisões sobre o que fazer com cada *email* de maneira a pressionar o menor número de teclas uma vez que existem milhares de *emails* a regularizar.

6.1 Operações *SQL* principais

As decisões sobre o que fazer com cada *email* foram apenas a combinação de duas operações básicas na base de dados:

- A desativação de um registo na tabela *email*.
- A duplicação de um registo na tabela, mais as suas associações, com um endereço de *email* diferente.

É do interesse da empresa desativar um *email,* mas mantê-lo na base de dados de modo a não perder a sua informação e saber que não o deve contactar outra vez.

Quanto à duplicação de registos de *emails* com um endereço novo, a intenção é atualizar a informação na BD, mas manter a informação anterior, como se fosse um histórico de atualizações do endereço. De acordo com a especificação, a duplicação era uma cópia integral de um registo na tabela *email* para a mesma tabela (sem esquecer as suas associações às restantes tabelas-entidade), com apenas o *ID*, as datas de modificação/criação e o endereço de *email* modificados.

Existem métodos mais adequados para guardar o histórico de endereços de *email* de uma empresa, como por exemplo, guardar os endereços anteriores à parte em uma tabela associada. Por outro lado, este método requer alterações adicionais na arquitetura da BD. Por falta de tempo e de privilégios na minha conta de utilizador da BD, não tive oportunidade para implementar dessa maneira.

À data de escrita deste relatório de estágio, a BD de *marketing* ainda não sofreu as alterações à arquitetura planeadas na secção 5.6. Por este motivo, o código usado nos exemplos ainda opera sobre o estado original da BD.

Desativação de um registo de *email*

Para desativar um registo de *email*, na tabela *email*, é necessário atualizar três campos: a data de modificação para a data/hora corrente, o campo de inativo, que funciona como uma variável booleana, para verdadeiro, e opcionalmente, uma causa da desativação do registo, como descrito na listagem 22.

Listagem 22: Comando *SQL* para desativação de um registo de email

```
UPDATE email SET (date_modified, inactive, inactive_cause)
= (current_timestamp, true,      %s)
WHERE id=%s;
```

Duplicação de um registo de *email*

Para duplicar um registo de *email*, basta fazer uma inserção a partir da seleção do registo original, com alguns campos alterados: a data de criação/modificação do registo, e o endereço novo, como exemplifico na listagem 23.

Listagem 23: Comando *SQL* para a duplicação de um registo de *email*

```
INSERT INTO email(id, date_created, date_modified, actor_legal_name, country,
cep, simple_nif, isic, identifier, simple_code, email, user_name,
last_date_sent, inactive, inactive_cause)

SELECT nextval('email_id_seq'), current_timestamp, current_timestamp,  ac-
tor_legal_name, country, cep, simple_nif, isic, identifier, simple_code, %s,
user_name, last_date_sent, false, NULL
FROM email WHERE id=%s
RETURNING *;
```

As associações do registo original também são duplicadas. Para cada tabela de associação, duplica-se as relações com *ID* do *email* original, mas usando o *ID* do registo do novo endereço *email*. Esta operação é análoga em todas as tabelas de associação, por isso, apenas exemplifico para o caso *email_origin*, na listagem 24.

Listagem 24: Comando *SQL* para a duplicação de um registo de *email*

```
INSERT INTO email_origin(id, email_id, origin_id)
SELECT nextval('email_origin_id_seq'), %s, origin_id
FROM email_origin WHERE email_id=%s
RETURNING *;
```

6.2 Fluxo de operações do programa

Este programa, como habitual, começa por ler um ficheiro de entrada, folha de cálculo, mais outro ficheiro com credenciais para se ligar à base de dados.

De seguida, o utilizador pode lançar um processo que desativa automaticamente *emails* na base de dados de *marketing*. Este processo não requer nenhuma interação por parte do utilizador pois opera sobre *emails* que garantidamente devem ser desativados da BD.

Alternativamente, o utilizador pode lançar o processo de tratamento manual de *emails* com classificações distintas. As várias classificações de *emails* são: *emails* que possuem códigos de erro, *emails* que são respostas por humanos, *emails* que são respostas automáticas, *emails* com endereços que não existem (provavelmente mal escritos) e *emails* que acusam serem *SPAM*.

No tratamento manual, para cada tipo de classificação é oferecido ao utilizador a informação relativa ao *email*, a possível causa da desativação (proveniente do ficheiro de entrada) e novos endereços de *email* sugeridos para adicionar à base de dados. O utilizador deve decidir se quer manter ou desativar o endereço de *email* original, e adicionar os endereços de *email* sugeridos.

As figuras 5 e 6 exemplificam o menu de processamento manual e um exemplo de disposição da informação de um *email*.

Para *emails* que não existem (possivelmente mal escritos) é oferecida ao utilizador a opção para corrigir o endereço do *email*. No lado da base de dados, isto significa que o *email* original é desativado e o novo endereço é duplicado a partir do original.

De acordo com a especificação, foi necessário criar opções para cada combinação de casos de modo a minimizar o número de teclas pressionadas pelo utilizador. As várias opções são:

- Desativar o *email* original, apenas.
- Desativar o *email* original e adicionar todos os *emails* novos sugeridos.
- Desativar o *email* original e adicionar alguns dos *emails* novos sugeridos.
- Manter o *email* original e adicionar todos os *emails* novos sugeridos.
- Manter o *email* original e adicionar alguns dos *emails* novos sugeridos.
- Ignorar e passar à frente.
- Renomear o *email* original.

6.3 Versão experimental com interface gráfica

O programa de regularização descrito neste capítulo foi desenvolvido em paralelo com uma outra versão do mesmo programa que usa uma interface gráfica, usando a biblioteca standard do *Python*, *tkinter*, [19].

Como nunca tive formação em *design* de interfaces, nem investiguei nada no tópico para além da montagem básica, o resultado final foi que se esperaria, ver figura 7. No entanto é um tópico interessante que gostaria de melhorar no futuro, principalmente pelas vantagens que proporciona.

A estrutura do programa ficou em algumas partes mais complicada, pelo elevado número de *widgets* que tive de configurar e posicionar. Os *widgets* são os elementos nas janelas, como por exemplo, botões, etiquetas, caixas de texto, listas. Por outro lado, o programa ficou mais simples dando a possibilidade de configurar as combinações das ações a realizar sobre um *email*, numa só janela.

Outras vantagens de usar uma interface gráfica são:

- Moldar o texto, o que não pode ser (trivialmente) feito em consola.
- Usar várias fontes (estilo de letra) para uma leitura mais confortável para o utilizador.
- Salientar palavras por cores, embora seja possível imprimir a cores na consola, requer mais esforço para implementar.
- Corrigir o endereço de *email* original numa entrada pré-preenchida.

- Executar em paralelo processamento automático de *emails* e o processamento manual, bem como a possibilidade de atualizar uma barra de progresso em tempo real.

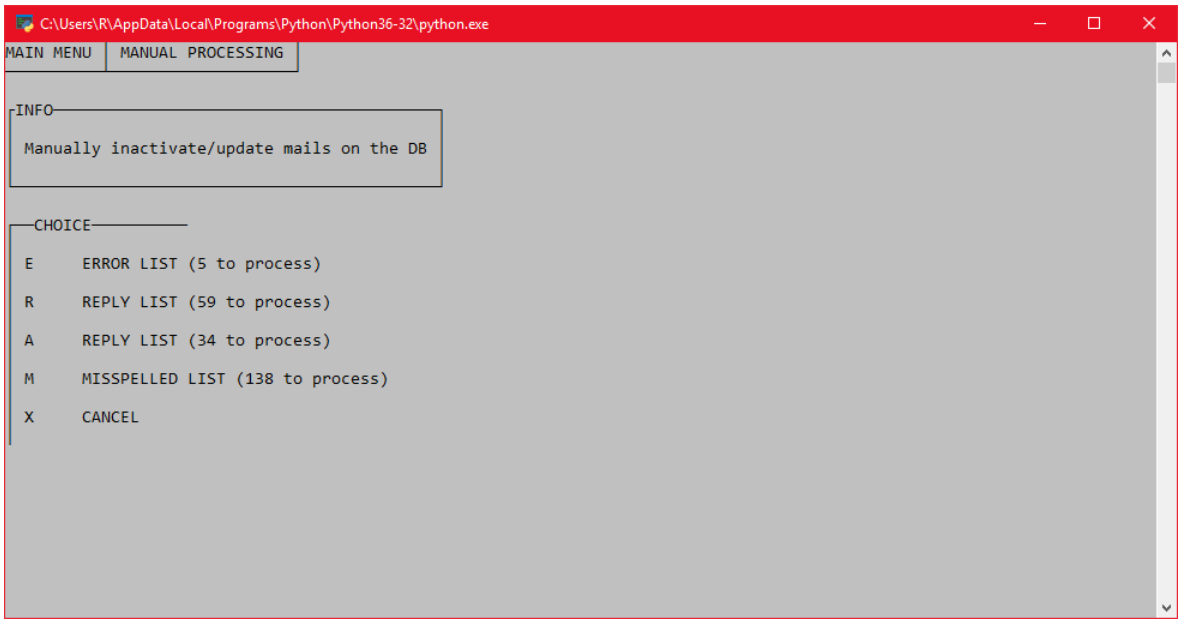

Figura 5: Interface de consola do programa (menu de processamento manual)

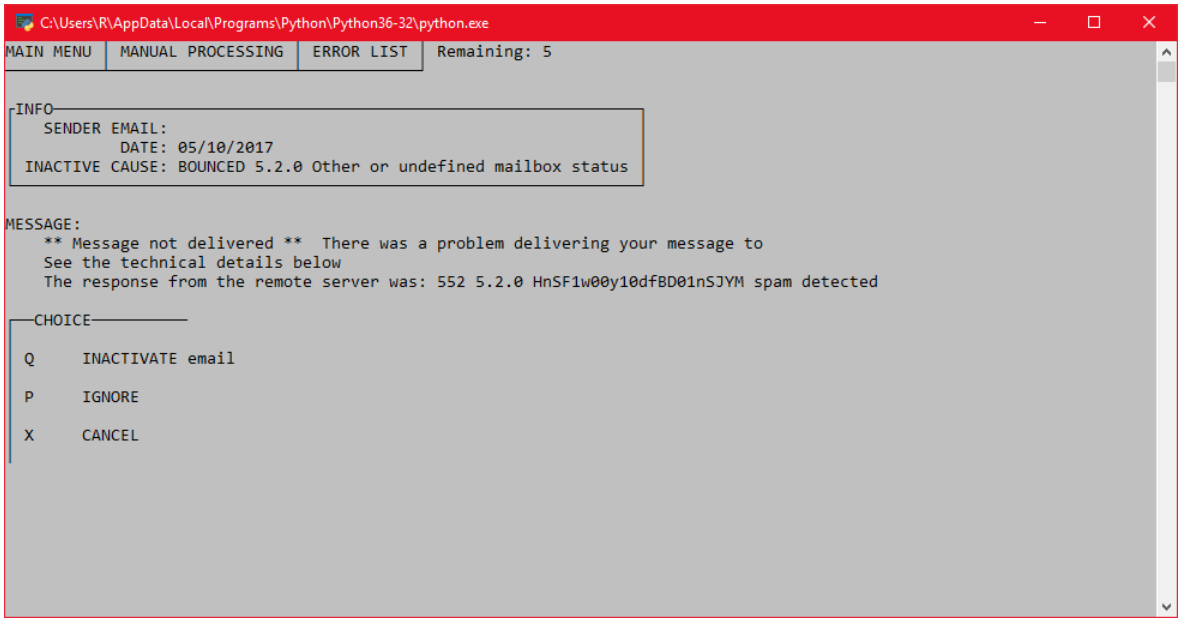

Figura 6: Interface de consola do programa (*emails* com erros)

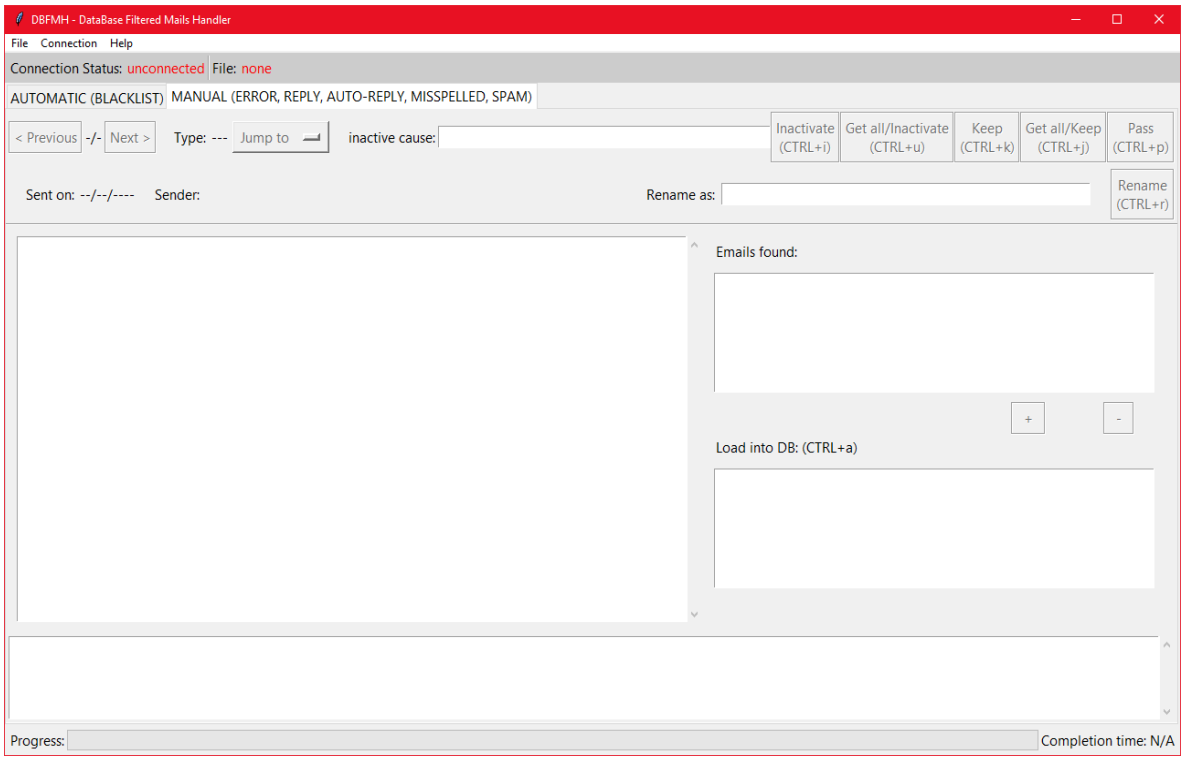

Figura 7: Interface de consola do programa (aba de processamento manual)

7. Extra

As tarefas incluídas neste capítulo estão relacionadas com atividades que desenvolvi na centroProduto que não foram suficientemente extensas para serem consideradas como capítulos ou então foram de natureza experimental.

7.1 Propostas de tratamento de dados em massa: *keywords*

Esta secção está relacionada com a reorganização das palavras-chave da secção 5.4. Na tabela *keyword* da BD de *marketing*, além de existirem registos de listas de palavras-chave em texto, também existem registos de texto que descrevem brevemente a empresa através de frases com estrutura. O texto descritivo da empresa ao ser reorganizado pelo programa do capítulo 5.4, que separa a lista (como texto) em registos separados, seria separado em demasiados registos de palavras que não tem relevância como palavra-chave.

Este problema relaciona-se com o tópico de processamento de linguagem natural (*NLP*), abordado em disciplinas do meu mestrado (*Data Mining*, Sistemas Inteligentes e Seminário de Computação). Por iniciativa própria, já tinha pesquisado acerca de bibliotecas para *Python* sobre o tópico *NLP*:

- *RAKE* (*Rapid Automatic Keyword Extraction*), [20], que analisa textos e extrai as *keywords* que considera mais relevantes.
- *TextBlob*, [21], que consegue identificar as classes de palavras, [22].

O *RAKE* consegue extrair composições de *keywords*, isto é, palavras-chave não singulares, mas necessita de um texto extenso para obter resultados suficientemente bons. Os textos descritivos das empresas estão guardados em *strings* com comprimento de 256 caracteres, por isso pode não ser a ferramenta ideal.

A biblioteca *TextBlob* é capaz de identificar classes de palavras. As classes de palavras organizam as palavras de uma língua, como o próprio nome indica, em classes. Exemplos destas classes são:

- Substantivos, que são os nomes de seres vivos, objetos, sentimentos, conceitos, etc… (Por exemplo: empresa, comércio, negócio, serviços)

- Adjetivos, que atribuem características aos substantivos. (Por exemplo: competente, eficaz)
- Verbos, que descrevem ações. (Por exemplo: comprar, vender)
- Pronomes, que determinam uma pessoa. (Por exemplo: eu, você, aquilo)
- ...entre outros.

A maior parte das palavras-chave têm tendência a serem substantivos ou adjetivos seguidos de substantivos. A ideia que propus foi de extrair apenas estas classes de palavras do texto usando a biblioteca *TextBlob*. Este método simples (que usa uma ferramenta complexa) parece ser bastante eficaz de acordo com os pequenos testes realizados, especialmente se a alternativa for destacar alguém para fazer a extração manual.

Tanto o *TextBlob* como o *RAKE* funcionam apenas em textos da língua Inglesa, no entanto, existem ferramentas eficientes que permitem detetar a língua a partir de um excerto de texto e traduzi-lo para Inglês. O *googletrans*, [23], é uma das bibliotecas em *Python* capaz de traduzir texto com resultados excelentes, mas possui uma restrição para a quantidade de texto que pode traduzir por dia.

No método que usa a biblioteca *TextBlob* existe alguma conveniência em operar sobre uma língua única (o Inglês) porque não é necessário adaptar o algoritmo à variação da estrutura das frases, que pode variar consoante a língua. Por exemplo, em Inglês os adjetivos precedem os substantivos ("*A red apple.*") mas em Português, os adjetivos sucedem os substantivos ("Uma maçã vermelha").

Para apresentar e demonstrar as propostas, desenvolvi uma interface gráfica simples que salienta as palavras-chave em uma caixa de texto e dispõe a informação resultante noutra caixa de texto, conforme ilustrado na figura 8. O *RAKE*, por exemplo, devolve um *ranking* de palavras-chave. Quanto maior for o ranking, maior é a probabilidade de ser uma palavra-chave relevante. O *TextBlob*, por outro lado, devolve a lista de palavras com a sua classe associada.

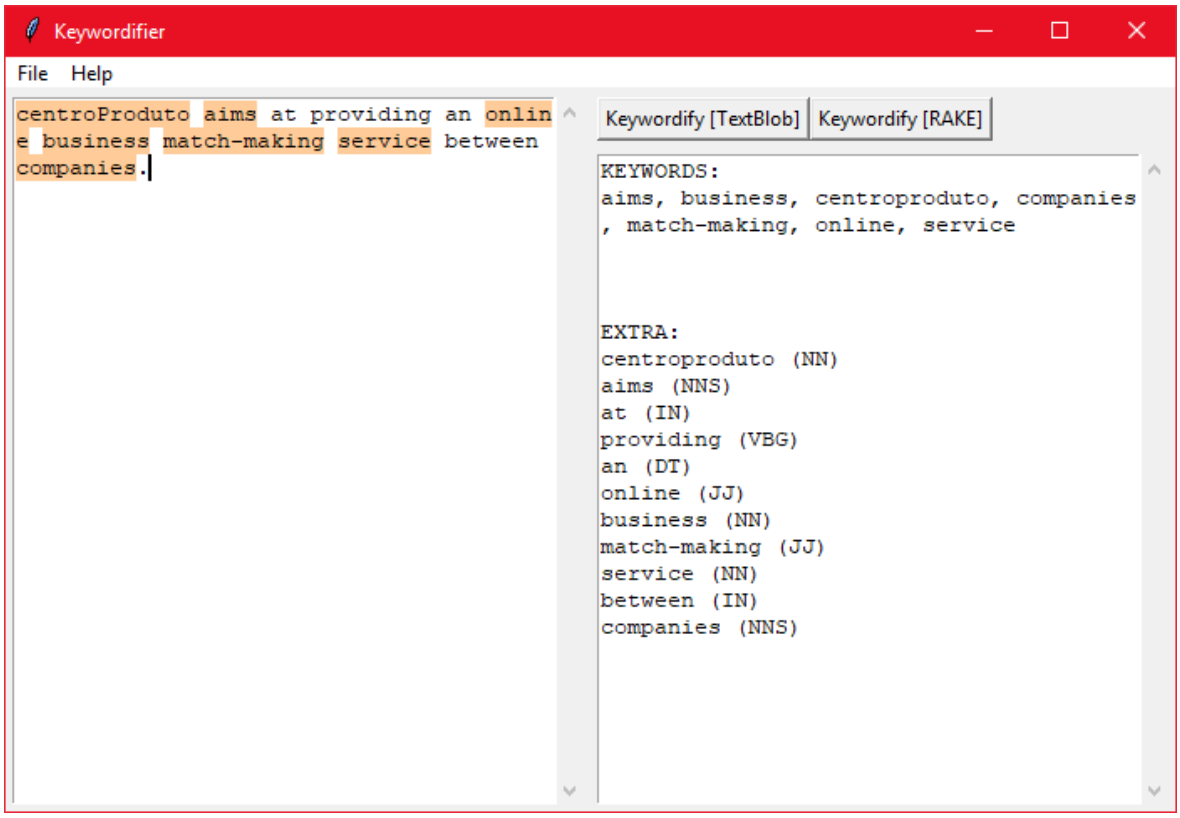

Figura 8: Interface demonstrativa da extração de *keywords*

A motivação que tive em investigar este tópico e apresentar as bibliotecas à empresa prendeu-se com o potencial que estas têm para resolver problemas relacionados com a análise de texto, especialmente se sua solução evitar que membros da empresa sejam sujeitos a tarefas bastante monótonas.

7.2 Classificação de *emails* retornados

A tarefa desta secção consistiu em desenvolver um programa que classifica uma série de *emails* retornados em consequência de uma campanha de *marketing* pela centroProduto. A classificação serve para identificar endereços de *email* que devem ser removidos dos contactos. Adicionalmente, o programa tenta adquirir novos endereços a partir de *emails* classificados como respostas. De acordo com a especificação, a condição mais importante exige que o programa não deva classificar incorretamente *emails* para serem removidos.

O programa percorre um ficheiro de texto que contém uma série de *emails*, separando cada um pelo cabeçalho, que tem um formato único. De seguida, o programa extrai al-

guma informação relevante a partir do *email*: o recipiente, o remetente, a data de envio e com alguma dificuldade extrai o corpo da mensagem.

Para classificar o *email*, o programa procura por códigos de erro, [24], e declarações de autorresposta ou de *SPAM*, entre outros indicadores únicos, que os servidores (por onde as mensagens circulam) introduzem nos cabeçalhos. Para isto, o programa serve-se de expressões regulares para obter os códigos de erro que têm um formato próprio: três dígitos separados por dois pontos.

Os códigos de erro têm uma causa associada, como por exemplo, a caixa de correio estar cheia, o *email* ter sido identificado como *SPAM* ou o recipiente não existir. A análise dos códigos de erro é útil para dar à centroProduto a informação necessária para adaptar a estratégia de *marketing*.

Um *email* retornado por causa da caixa de correio estar cheia significa que existe uma probabilidade alta de ter entrado em desuso pelo que não interessa continuar a ser contatado pela centroProduto.

Emails retidos por terem sido identificados como *SPAM* sugerem uma alteração no fraseamento das mensagens e nas vias como a centroProduto contacta as empresas.

Se um *email* vier retornado porque o recipiente não existe então é porque endereço provavelmente está mal escrito.

O programa consegue também identificar autorrespostas, que são programadas por pessoas para informar sobre o estado da empresa ou do endereço em si. Grande parte das autorrespostas orientam os remetentes para contactar as empresas através de novos endereços de email, que o programa depois extrai.

Um ficheiro cedido para o desenvolvimento do programa com pouco mais do que 1100 *emails* referentes a um período de cerca de um mês, tinha cerca de 126MB. Este volume aumenta rapidamente quando as mensagens contêm conteúdos multimédia, como imagens dos logotipos das empresas. Como medida de otimização, o programa ignora o que não é cabeçalho ou blocos de texto-pleno ou *HTML* nos *emails*.

7.3 Melhoramentos na experiência do utilizador

Esta secção detalha algumas medidas aplicadas para melhorar a experiência do utilizador nos programas desenvolvidos.

Carregamento de teclas nos menus

Originalmente, os programas pediam ao utilizador que escrevesse a letra correspondente à opção e depois carregasse a tecla *ENTER* para submeter. Para tornar mais prática a forma como os utilizadores interagem com os programas, desenvolvi um mecanismo que faz com que as opções sejam diretamente selecionadas através do carregamento de teclas. Esta medida é particularmente útil quando os programas frequentemente apresentam menus.

Através da biblioteca standard do *Python msvcrt* é possível aceder a todas as teclas que foram carregadas, que ficam guardadas num *buffer* e manipulá-lo através da programação para desenvolver um mecanismo que apanha a tecla carregada. Um *buffer* pode ser visto como uma de fila de espera que guarda dados temporariamente, por ordem de chegada.

A listagem 25 descreve o processo de leitura de carregamento de teclas. Quando a função *get_keypress()* é chamada para ler um carregamento de tecla, começa por esvaziar o *buffer*, caso contrário o programa iria ler a tecla pressionada mais antiga armazenada no *buffer*. De seguida espera pelo carregamento de tecla e devolve a tecla.

Listagem 25: Leitura do carregamento de uma tecla

```
def get_keypress():

      while msvcrt.kbhit(): msvcrt.getch()
      ans = msvcrt.getch()

      #treat ans input

      return ans
```

Aspetos visuais da consola

Desenvolvi um pequeno módulo, exposto no apêndice, que serve para dispor informação na consola de maneira mais agradável ao olhar. Este módulo suporta três funções de impressão e uma classe que define cores de fundo e palavras. As funções incluem a impressão de:

- mensagens em caixa, com a possibilidade de colocar um título, escolher o estilo da moldura e dar margens horizontais/verticais.
- barra de progresso que serve para representar a evolução de um processo.
- arvore de listas, que imprime listas (com sub-listas) em arvore hierárquica.

Listagem de ficheiros

Quase todos os programas desenvolvidos leem ficheiros de entrada. Originalmente os programas recebiam os nomes dos ficheiros de entrada escritos manualmente na consola, o que implicava memorizar ou verificar o nome do ficheiro. Caso o utilizador se enganasse a escrever, teria de repetir este passo.

Para facilitar a seleção do ficheiro de entrada, desenvolvi uma função que lista os ficheiros de um diretório, que usualmente é aquele onde o programa é executado. É também possível filtrar a lista dos nomes dos ficheiros pela extensão desejada de modo a minimizar o número de opções apresentado ao utilizador. A listagem 26 mostra como aplicar este método recorrendo à biblioteca integrada *os* (*Operating System*), [25].

Listagem 26: Obtenção da lista de ficheiros de um diretório

```
def get_current_directory():
	return os.path.dirname(os.path.realpath(__file__))

def get_list_files_from_directory(directory):
	current_directory_files = [f for f in os.listdir(directory) if
os.path.isfile(os.path.join(directory, f))]
	return current_directory_files

def filter_list_files_with_extention(list_of_files, extention):
	if not extension.startswith('.'): extension = '.' + extension
	list_of_files_with_given_extention = [f for f in list_of_files if
f.endswith(extension)]
	return list_of_files_with_given_extention
```

Estimador temporal

Os programas que executam processos longos têm um estimador temporal que dita o tempo que falta e a data/hora prevista para conclusão.

O tempo que falta para terminar um processo é calculado pela média de tempo de cada elemento processado multiplicado pelos elementos que faltam processar. A média de tempo de cada elemento é calculado pelo tempo total que passou desde o início do processo a dividir pelo número de elementos processados. A data/hora prevista de conclusão é a soma do tempo atual com o tempo estimado, convertida no formato de data.

$$remainingTime = (totalToProcess - totalProcessed) * averageProcessTime$$

$$averageProcessTime = elapsedTime / totalProcessed$$

Operações com folhas de cálculo

A biblioteca para *Python*, *openpyxl*, oferece mecanismos para operar sobre folhas de cálculo compatíveis com o *Excel* (*Microsoft Office*) e o *Google Drive*, usados pela empresa.

Na centroProduto, o formato usado para registar praticamente todo a informação é a folha de cálculo. Os dados de entrada e saída dos programas não são exceção. O uso das folhas de cálculo deve-se principalmente à maneira como estas permitem apresentar os dados, que têm por vezes com estruturas mais complexas.

Por outro lado, é possível ler e escrever em pontos específicos dos ficheiros de folha de cálculo, o que dá alguma segurança na manipulação dos dados. Uma técnica usada para ajudar neste ponto consiste em arranjar um dicionário que faz uma bijeção entre o nome das colunas de dados que o programa aceita e as colunas onde estas se localizam. Outra técnica consiste em associar a cada registo processado um atributo que indica a posição da linha da folha de onde foi lido para posteriormente ser possível escrever na mesma posição.

Nas tarefas realizadas nunca foi necessário fazer muito mais do que ler, escrever e guardar os ficheiros, no entanto houve necessidade de implementar algumas medidas para

diminuir o atraso causado por estas operações. Quando as folhas de cálculo possuem um grande volume, as operações de leitura e escrita demoram minutos a concluir, mesmo em máquinas boas. Este problema é agravado quando os utilizadores introduzem alguma formatação nos ficheiros.

Para contornar este problema, por vezes é necessário ou guardar a informação em ficheiros temporários simples ou guardar por intervalos.

7.4 Interfaces gráficas e *threading*

Nos intervalos entre tarefas aproveitei para investigar acerca de interfaces gráficas com a intenção de melhorar a experiência do utilizador. O *Python* possui uma biblioteca standard chamada *tkinter* que permite construir interfaces gráficas sem que o utilizador necessite de instalar nada.

O *tkinter* permite criar e configurar janelas e compor a sua interface através de *widgets*. Os *widgets* são funcionalidades para interagir com a janela e dispor informação. Opcionalmente, é possível definir *frames* (áreas retangulares) onde se podem inserir *widgets* e mais *frames*. Após definir os *frames* e as *widgets*, é necessário posicioná-los na janela. O *tkinter* oferece três maneiras de o fazer: por coordenadas, por grelha, ou por posicionamento automático.

Fui capaz de desenvolver algumas demonstrações e programas auxiliares com interfaces bastante simples, no entanto, existia um problema que impedia o seu uso nas tarefas de processamento. Programas que executavam passos demorados congelavam as janelas até terminarem. Este é um problema que um desenvolvedor principiante nas interfaces gráficas se depara frequentemente. Para conseguir ultrapassar este obstáculo, foi necessário mais algum tempo de investigação acerca de outro tópico, *threading*.

O *threading* permite correr em paralelo pedaços de código de um programa. Neste caso, o pretendido era correr a interface num *thread* e correr o código que processava os dados noutro *thread* separado. O *Python* oferece uma biblioteca standard chamada *threading*, [26], para este fim. Admito que este tópico foi difícil para mim, não só pela falta de bases como também pelo pouco tempo que tive disponível. Apenas na parte final do estágio

consegui desenvolver exemplos de programas com interfaces gráficas e processamento de dados em paralelo.

O tópico de *threading* levou-me também a investigar o multiprocessamento (*multi processes*), isto é, múltiplos processos que executam em simultâneo, se o hardware o permitir. No futuro, este tópico poderá permitir acelerar bastante algumas tarefas cujas partes do processamento dependem de recursos distintos da máquina.

Por exemplo, se o processo de um elemento depende da ligação à internet, da escrita em disco e de cálculos, então poderá ser possível processar vários elementos em simultâneo enquanto estes usam recursos diferentes da máquina.

7.5 Manutenção dos programas

Parte do meu papel como programador na empresa envolveu a manutenção de programas. A manutenção de programas consiste em fazer pequenos ajustes para corrigir erros, adições às suas funcionalidades, bem como repará-los quando surgem problemas de compatibilidade.

Devido à experiência que tive logo na primeira tarefa de programação realizada, que necessitou de muitos ajustes, percebi que era útil fazer código organizado e modular de maneira a que as alterações futuras não levassem demasiado tempo a implementar.

Por exemplo, passado cerca de três meses de ter sido desenvolvido e um mês depois do lançamento no portal, o módulo *FASTINSERT()*, do capítulo 3, necessitou de uma atualização para aceitar mais um campo. O campo adicionado foi uma *URL* personalizada da página-perfil da empresa.

Neste caso, adicionar um campo novo no módulo implicou os seguintes passos que se resumem a uma linha de código em pontos localizados:

- Adicionar o campo a uma lista de nomes de campos válidos na fase de processamento de linguagem.
- Por ser obrigatório, na fase de análise de requisitos adicionar uma condição para testar a sua existência.

- Como a *URL* também requer controlo de qualidade, adicionar outra função de validação, providenciada mais uma vez pela centroProduto.
- Adicionar o campo à função de *aliasing*, que renomeia os campos.
- Adicionar o campo à composição do dicionário de resposta.

O programa de envio de mensagens no portal centroProduto, do capítulo 4, foi outro exemplo de um programa que teve de ser reparado, mais do que uma vez. As atualizações do portal centroProduto alteraram o código fonte fazendo com que o programa deixasse de conseguir encontrar os botões e formulários. A atualização do *Google Chrome* fez com que o *webdriver* deixasse de funcionar nas máquinas dos utilizadores.

Neste caso, bastou apenas corrigir os caminhos para os elementos da página no programa e fazer uma atualização a uma instrução para usar uma versão mais recente do *webdriver*. Para ajudar na manutenção deste programa durante a minha ausência na empresa, deixei as *strings* das referências dos elementos das páginas para que qualquer outro programador da empresa os pudesse reparar facilmente, sem ter de perder tempo a ler o resto do código.

8. Conclusão

8.1 Impacto das tarefas realizadas na operação de *marketing*

As tarefas por mim realizadas neste estágio enquadram-se numa operação de *marketing* realizada pela centroProduto. O meu plano de atividades deste estágio consistiu em usar a programação para automatizar e acelerar esta operação. Neste capítulo descrevo como os programas desenvolvidos auxiliaram a empresa pela ordem da operação de marketing, como resumida na tabela 8.

O programa de *web scraping* do capítulo 2 ajudou a completar dados sobre as empresas. Entre os dados extraídos, creio que os dados referentes à atividade empresarial ajudaram a orientar as campanhas de *marketing* especializadas por setores comerciais. O programa também extraía o NIF e o código postal o que permitiu preparar os dados para uma posterior inscrição mais rápida no portal.

Os dados extraídos pelo programa de *web scraping*, bem como dados obtidos de outras fontes tiveram de ser importados para a BD de *marketing* da empresa pelo programa desenvolvido na secção 5.3. Devido ao desempenho no desenvolvimento do programa de importação, foram-me também atribuídas tarefas de administração da BD para alterar a sua arquitetura e corrigir erros causados previamente.

A empresa serviu-se dos dados organizados na BD para lançar várias campanhas de *marketing* o que originou um aumento de clientes no portal. Muitos dos clientes recém-chegados deparavam-se com dificuldades na inscrição e contactavam a empresa para lhes dar suporte. Para responder a esta necessidade, a empresa desenvolveu uma funcionalidade chamada *FASTINSERT()* para criar, de maneira mais prática, contas de utilizadores com empresas associadas. O módulo desenvolvido no capítulo 3 foi integrado nesta funcionalidade.

O programa desenvolvido no capítulo 4 serviu para fins de *marketing* dentro do portal da centroProduto. O *marketing* neste caso foi maioritariamente direcionado a novos utilizadores do portal.

O programa de classificação de *emails* na secção 7.2 em conjunção com o programa de regularização da BD do capítulo 6 auxiliaram na filtragem de *emails* obsoletos para *marketing*, bem como na extração de *emails* para serem usados em campanhas de *marketing* futuras.

Devo notar que a operação de *marketing* foi auxiliada por programas que criei ou melhorei, bem como programas desenvolvidos por outros membros. Por este facto, as tarefas desenvolvidas obedecem a uma ordem de prioridades que dependiam das necessidades que iam surgindo. Adicionalmente, as tarefas realizadas estão em concordância com o plano de estágio acordado entre a empresa e a Universidade de Aveiro.

Capítulo	**Descrição**
	Recolha de dados de empresas a partir de bases de dados de *websites*, realizado por outro colega.
2	Recolha de dados adicionais de empresas (como o NIF, o CAE, o código-postal…)
5.3	Inserção dos dados na base de dados de *marketing*.
	Envio de *emails* para as empresas na base de dados de *marketing*, realizado pela equipa de *marketing*.
3	Desenvolvimento de um módulo de uma funcionalidade que permite a equipa de *marketing* inserir empresas no portal com maior facilidade. Em consequência do passo anterior.
4	Desenvolvimento de um programa de envio de mensagens pelo portal, destinado em grande parte, a empresas recentemente registadas.
Extra	Classificação de *emails* retornados, para eliminar os obsoletos da base de dados, e tentar contactar as empresas com os *emails* novos. Desenvolvido em parceria com outro colega.
6	Atualização da base de dados através da lista resultante do programa do passo anterior.

Tabela 8: Resumo de operação de *marketing*

8.2 Desenvolvimento das técnicas de programação

As tarefas que realizei no estágio centraram-se no desenvolvimento de programas. O meu dia a dia na empresa consistiu em escrever código para resolver os vários problemas das tarefas. Como consequência, as minhas técnicas de programação evoluíram consideravelmente, sem com isto querer implicar que passei a dominar a arte.

Existem alguns aspetos que tive de melhorar, mais relacionados com técnicas e práticas de programação do que com as tarefas atribuídas, já que os programas desenvolvidos frequentemente ultrapassavam as mil linhas de código.

À medida que os programas eram apresentados para testes e posterior aplicação, era comum receber instruções para fazer pequenos melhoramentos e adicionar funcionalidades extra. Por vezes, os pequenos melhoramentos passavam por fazer ligeiros retoques no código, como corrigir erros ortográficos ou editar elementos gráficos na interface. Estes pequenos melhoramentos podiam também obrigar a uma alteração ao longo da estrutura ao longo código, sendo a integração de contagens estatísticas um exemplo.

Para combater estas situações tentei o quanto possível escrever o meu código claro, organizado, conciso e modular. O *Python* desde logo obriga a que o código esteja devidamente indentado, no entanto, cabe ao programador nomear os objetos e funções de tal modo que os seus propósitos sejam evidentes.

O código organizado: por blocos ou com comentários devidamente colocados ajuda a ter uma melhor perceção da sua estrutura.

Além das técnicas de programação que fui desenvolvendo gradualmente, foi também necessário aprender a usar uma diversidade de bibliotecas em *Python* como ferramentas de auxílio às minhas tarefas. Considero que este ponto me tornou mais autónomo e influenciou a procurar e explorar outras bibliotecas, como foi o caso da proposta de tratamento dos dados na secção 7.1. As diversas capacidades adquiridas e desenvolvidas durante este estágio, relacionadas com o uso de bibliotecas incluem:

- Leitura, escrita e manipulação de ficheiros com diversos formatos como por exemplo folhas de cálculo.
- Extração informação e interação com *websites*.

- Tratamento de texto através de expressões regulares.
- Interação com BD.
- Processamento de linguagem natural.
- Criação de interfaces gráficas.
- Paralelização de tarefas.

Não só fiquei satisfeito com as competências adquiridas como também motivado para continuar a melhorar mais e mais como programador.

Terminologia

backup: Cópia de segurança.

BD (Base de Dados): Coleção organizada de dados que podem ser acedidos e manipulados.

BeautifulSoup4: Biblioteca disponível em *Python* para percorrer facilmente um objeto com estrutura *XML*.

CAE (Classificação das Atividades Económicas): Código que classifica as atividades económicas de uma empresa. Qualquer empresa é obrigada por lei a ser classificada com pelo menos um código de atividade. É possível que uma empresa possua vários códigos de atividade consoante os serviços que presta ou os bens que produz.

caso de uso: Funcionalidade de um sistema.

CEP (Código de Endereçamento Postal): Código que identifica um espaço geográfico.

difflib: Biblioteca em *Python* que oferece ferramentas para comparar sequências.

ID: Identidade, identificador, atributo de um registo (ou elemento) que o define unicamente numa coleção de dados.

login: Ato de entrar num sistema através da verificação das credências de uma conta de utilizador.

NIF (Número de Identificação Fiscal): Código que identifica uma entidade fiscal em atividades financeiras ou impostos.

openpyxl: Biblioteca em *Python* para manipular ficheiros folha de cálculo.

palavra-chave: *String* que etiqueta um elemento ao qual está associada.

pgAdmin 4: Aplicação com interface gráfica para interação com bases de dados em *PostgreSQL*.

pop-up: Janela/formulário que aparece no meio do ecrã de maneira a chamar a atenção do utilizador.

PostgreSQL: Sistema de gestão de bases de dados que usa a linguagem *SQL*.

psycopg2: Biblioteca em *Python* que permite estabelecer ligações a bases de dados e executar comandos de manipulação de dados.

Python: Linguagem de programação. Destaca-se das demais linguagens pela ênfase que deposita na leitura fácil do código através do uso cuidado de indentações.

re, REGEX (REGular EXpressions): Expressões Regulares. Expressões que definem padrões de *strings*.

requests: Biblioteca em *Python* que permite descarregar o conteúdo de uma página de um *website*.

SGBD (Sistema de Gestão de Bases de Dados): *Software* que permite definir, consultar, manipular e administrar os dados de uma base de dados.

SQL (Structured Query Language): Linguagem standard para operar sobre bases de dados relacionais.

string: Tipo de dados que representa uma sequência de caracteres.

thread, **threading**: Linha de processamento, método de um programa dividir a sua execução em sub-tarefas que podem ser executadas simultaneamente.

tkinter: Biblioteca standard em Python para composição de interfaces gráficas.

URL (Uniform Resource Locator): Endereço de uma página de um *website*.

web browser: Aplicação para aceder e interagir com *websites*.

Web Scraping: Extração de informação a partir de *websites*.

webdriver (*Selenium* webdriver): Aplicação que simula a interação de um utilizador com um *web browser*.

website: Sítio da *internet*.

XML (*eXtensible Markup Language*): Linguagem de marcação desenhada para guardar informação estruturada para leitura percetível tanto por seres-humanos como por computadores.

Referências

1 – Apresentação da empresa no portal
https://www.centroproduto.com/#/en/--about-us
(Acedido a 31/05/2018)
2 – Informação sobre a empresas na sua página do Facebook
https://pt-pt.facebook.com/pg/centroproduto.pt/about/?ref=page_internal
(Acedido a 31/05/2018)
3 – Artigo do Jornal de Notícias
http://www.jornaldenegocios.pt/empresas/detalhe/a_rede_social_que_liga_as_pme_ao_mundo
(Acedido a 31/05/2018)
4 - Documentação da biblioteca openpyxl:
https://openpyxl.readthedocs.io/en/2.5/usage.html
(Acedido a 08/10/2017)
5 - Documentação da biblioteca BeatifulSoup4
https://www.crummy.com/software/BeautifulSoup/bs4/doc/
(Acedido a 08/10/2017)
6 - Documentação (não oficial) da biblioteca Selenium
http://selenium-python.readthedocs.io/
(Acedido a 08/10/2017)
7 – Tutorial da biblioteca requests
http://docs.python-requests.org/en/master/
(Acedido a 08/10/2017)
8 – Documentação da biblioteca Python difflib:
https://docs.python.org/2/library/difflib.html
(Acedido a 25/04/2018)
9 – Gestalt Pattern Matching, Ratcliff/Obershelp (contexto)
http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1988/8807/8807c/8807c.htm
(Acedido a 27/04/2018)
10 – Gestalt Pattern Matching, Ratcliff/Obershelp (exemplo de aplicação)
https://ilyankou.files.wordpress.com/2015/06/ib-extended-essay.pdf
(Acedido a 27/04/2018)

11 – Gestalt Pattern Matching, Ratcliff/Obershelp (exemplo do código em Java)
https://github.com/GitSage/gestalt-pattern-matcher/blob/master/gestalt-pattern-matcher.js
(Acedido a 27/04/2018)
12 – Acerca do /robots.txt
http://www.robotstxt.org/robotstxt.html
(Acedido a 07/05/2018)
13 – Tipos de conteúdo das tags:
https://www.liquid-technologies.com/xml-schema-tutorial/xsd-elements-attributes
(Acedido a 29/05/2018)
14 – Tutorial da biblioteca re
https://www.tutorialspoint.com/python/python_reg_expressions.htm
(Acedido a 04/06/2018)
15 – PostgreSQL
https://www.postgresql.org/
(Acedido a 06/05/2018)
16 – Documentação da biblioteca psycopg2
http://initd.org/psycopg/docs/
(Acedido a 03/03/2018)
17 – Definição de base de dados
https://searchsqlserver.techtarget.com/definition/database
(Acedido a 16/05/2018)
18 – Normalização de bases de dados (definição, explicação)
https://www.studytonight.com/dbms/database-normalization.php
(Acedido a 17/0/2018)
19 – Tutorial da biblioteca tkinter
https://www.tutorialspoint.com/python3/python_gui_programming.htm
(Acedido a 04/06/2018)
20 – Tutorial da biblioteca RAKE
https://pypi.org/project/rake-nltk/
(Acedido a 03/04/2018)
21 – Tutorial da biblioteca TextBlob
http://textblob.readthedocs.io/en/dev/
(Acedido a 03/04/2018)
22 – Classes de palavras

https://www.infoescola.com/portugues/classes-de-palavras
(Acedido a 03/05/2018)
23 – Tutorial da biblioteca googletrans
https://pypi.org/project/googletrans/
(Acedido a 03/04/2018)
24 – Códigos de erro em emails:
https://www.iana.org/assignments/smtp-enhanced-status-codes/smtp-enhanced-status-codes.xhtml
(Acedido a 03/05/2018)
25 – Documentação da biblioteca os
https://docs.python.org/3/library/os.html
(Acedido a 04/06/2018)
26 – Tutorial da biblioteca threading
https://www.tutorialspoint.com/python/python_multithreading.htm
(Acedido a 04/06/2018)

Apêndice

Apêndice 1: Módulo de elementos gráficos para consola

```
#coding=utf-8

import os



##FORMAT
class formats():

    class style():
        BOLD = '\033[1m'
        UNDERLINED = '\033[4m'
        HIGHLIGHTED = '\033[7m'
        INVISIBLE = '\033[30m'
        COVERED = '\033[47m'

    #FONT
    class font():
        GREY = '\033[90m'
        RED = '\033[91m'
        GREEN = '\033[92m'
        YELLOW = '\033[93m'
        BLUE = '\033[94m'
        PURPLE = '\033[95m'
        CYAN = '\033[96m'
        WHITE = '\033[97m'

        GREY_DARK = '\033[30m'
        RED_DARK = '\033[31m'
        GREEN_DARK = '\033[32m'
        YELLOW_DARK = '\033[33m'
        BLUE_DARK = '\033[34m'
        PURPLE_DARK = '\033[35m'
        CYAN_DARK = '\033[36m'

    #BACKGROUND
    class background():
        GREY = '\033[100m'
        RED = '\033[101m'
        GREEN = '\033[102m'
        YELLOW = '\033[103m'
        BLUE = '\033[104m'
        PURPLE = '\033[105m'
        CYAN = '\033[106m'
        WHITE = '\033[107m'

        RED_DARK = '\033[41m'
        GREEN_DARK = '\033[42m'
        YELLOW_DARK = '\033[43m'
```

```
        BLUE_DARK = '\033[44m'
        PURPLE_DARK = '\033[45m'
        CYAN_DARK = '\033[46m'
        WHITE_DARK = '\033[47m'

    DEFAULT = '\033[0m'

##SUPPORT / OS
def echo_off():
    os.system("@echo off")

##BOXES
def print_box(message, **kwargs):

    ##ARGUMENTS
    #CONVERT MESSAGE INTO SENTENCES
    sentences = message.split('\n')

    #GET LONGEST SENTENCE LENGTH
    longest_sentence_length = 0
    for sentence in sentences:
        if len(sentence) > longest_sentence_length:
            longest_sentence_length = len(sentence)

    #DEFAULTS
    if 'title' in kwargs: title = kwargs['title']
    else:                 title = ''

    if 'style' in kwargs: style = kwargs['style']
    else:                 style = 'box'

    if 'hmargin' in kwargs: hmargin = kwargs['hmargin']
    else:                   hmargin = 0

    if 'vmargin' in kwargs: vmargin = kwargs['vmargin']
    else:                   vmargin = 0

    ##STYLE EDGE
    if style == 'box':
        bar_horizontal      = '─'
        bar_vertical        = '│'
        corner_top_left     = '┌'
        corner_top_right    = '┐'
        corner_bottom_left  = '└'
        corner_bottom_right = '┘'

    elif style == 'box_double':
        bar_horizontal      = '═'
        bar_vertical        = '║'
        corner_top_left     = '╔'
        corner_top_right    = '╗'
        corner_bottom_left  = '╚'
        corner_bottom_right = '╝'

    elif style == 'box_bold':
        bar_horizontal      = '━'
        bar_vertical        = '┃'
        corner_top_left     = '┏'
```

```
        corner_top_right    = '┐'
        corner_bottom_left  = '└'
        corner_bottom_right = '┘'

    elif style == 'round':
        bar_horizontal      = '─'
        bar_vertical        = '│'
        corner_top_left     = '╭'
        corner_top_right    = '╮'
        corner_bottom_left  = '╰'
        corner_bottom_right = '╯'

    elif style == 'dashed_2':
        bar_horizontal      = '╌'
        bar_vertical        = '╎'
        corner_top_left     = '┌'
        corner_top_right    = '┐'
        corner_bottom_left  = '└'
        corner_bottom_right = '┘'

    elif style == 'dashed_2_bold':
        "╌ ╌ ╎ ╎ ╌ ╌ ╎ ╎ "
        bar_horizontal      = '╍'
        bar_vertical        = '╏'
        corner_top_left     = '┏'
        corner_top_right    = '┓'
        corner_bottom_left  = '┗'
        corner_bottom_right = '┛'

    elif style == 'dashed_3':
        bar_horizontal      = '┄'
        bar_vertical        = '┆'
        corner_top_left     = '┌'
        corner_top_right    = '┐'
        corner_bottom_left  = '└'
        corner_bottom_right = '┘'

    elif style == 'dashed_3_bold':
        bar_horizontal      = '┅'
        bar_vertical        = '┇'
        corner_top_left     = '┏'
        corner_top_right    = '┓'
        corner_bottom_left  = '┗'
        corner_bottom_right = '┛'

    elif style == 'dashed_4':
        bar_horizontal      = '┈'
        bar_vertical        = '┊'
        corner_top_left     = '┌'
        corner_top_right    = '┐'
        corner_bottom_left  = '└'
        corner_bottom_right = '┘'

    elif style == 'dashed_4_bold':
```

```
        bar_horizontal      = '…'
        bar_vertical        = '┆'
        corner_top_left     = '┌'
        corner_top_right    = '┐'
        corner_bottom_left  = '└'
        corner_bottom_right = '┘'

    ##DRAW BOX
    #OFFSET WHEN TITLE IS LONGER THAN MESSAGE
    horizontal_margin_left_offset = 0
    if len(title) > (longest_sentence_length+2*hmargin):
        hmargin = (len(title)-longest_sentence_length)
        if hmargin % 2 == 1: horizontal_margin_left_offset = 1
        hmargin = hmargin//2

    #TOP
    print(corner_top_left, end='')
    print(title, end='')
    for i in range(2*hmargin+longest_sentence_length-len(title)):
print(bar_horizontal, end='')
    print(corner_top_right)

    #TOP-MIDDLE
    for i in range(vmargin):
        print(bar_vertical, end='')
        for j in
range(2*hmargin+horizontal_margin_left_offset+longest_sentence_length):
print(' ', end='')
        print(bar_vertical)

    #MIDDLE
    for sentence in sentences:
        print(bar_vertical, end='')
        for i in range(hmargin): print(' ', end='')
        print(sentence, end='')
        for i in range(longest_sentence_length-len(sentence)): print('
', end='')
        for i in range(hmargin+horizontal_margin_left_offset): print('
', end='')
        print(bar_vertical)

    #MIDDLE-BOTTOM
    for i in range(vmargin):
        print(bar_vertical, end='')
        for j in
range(2*hmargin+horizontal_margin_left_offset+longest_sentence_length):
print(' ', end='')
        print(bar_vertical)

    #BOTTOM
    print(corner_bottom_left, end='')
    for i in
range(2*hmargin+horizontal_margin_left_offset+longest_sentence_length):
print(bar_horizontal, end='')
    print(corner_bottom_right)

##PROGRESS BAR
def print_progress_bar(progress, total, **kwargs):
```

```
    #DEFAULTS
    if 'size' in kwargs: size = kwargs['size']
    else:                size = 100

    #EXCEPTIONS
    if progress < 0: return
    if total <= 0: return
    if size < 1: return

    try: size = int(size)
    except: return

    ##FRAME
    bar_horizontal      = '─'
    bar_vertical        = '│'
    corner_top_left     = '┌'
    corner_top_right    = '┐'
    corner_bottom_left  = '└'
    corner_bottom_right = '┘'

    ##DRAW
    #TOP
    print(corner_top_left, end='')
    for i in range(size+2): print(bar_horizontal, end='')
    print(corner_top_right)

    #MIDDLE
    print(bar_vertical+' ', end='')
    for i in range(round(size*(progress/total))):
print(formats.background.GREEN + ' ' + formats.DEFAULT, end='')
    for i in range(round(size*(1-progress/total))): print('░', end='')
    print(' '+bar_vertical)

    #BOTTOM
    print(corner_bottom_left, end='')
    for i in range(size+2): print(bar_horizontal, end='')
    print(corner_bottom_right)

##TREES
def print_tree(tree, **kwargs):

    def print_tree_aux(tree, branch_level, style, height_spacing):

        ##STYLE
        if style == 'line':
            I = '│'
            L = '└'
            LR = '└'
            T = '├'
            E = '-'
            R = '─'

        if style == 'bold':
            I = '┃'
            L = '┗'
            LR = '┗'
            T = '┣'
```

```
            E = '-'
            R = '━'

        if style == 'double':
            I = '║'
            L = '╚'
            LR = '╚'
            T = '╠'
            E = '-'
            R = '═'

        if style == 'dashed_2':
            I = '┆'
            L = '└'
            LR = '└'
            T = '├'
            E = '-'
            R = '--'

        if style == 'dashed_2_bold':
            I = '┆'
            L = '└'
            LR = '└'
            T = '├'
            E = '-'
            R = '--'

        if style == 'dashed_3':
            I = '┆'
            L = '└'
            LR = '└'
            T = '├'
            E = '-'
            R = '┈'

        if style == 'dashed_3_bold':
            I = '┆'
            L = '└'
            LR = '└'
            T = '├'
            E = '-'
            R = '┈'

        if style == 'dashed_4':
            I = '┆'
            L = '└'
            LR = '└'
            T = '├'
            E = '-'
            R = '┈'

        if style == 'dashed_4_bold':
            I = '┆'
            L = '└'
```

```
            LR = '└'
            T = '├'
            E = '-'
            R = '…'

        #ROOT
        print('■')

        ##DRAW
        for i in range(len(tree)):
            #HEIGHT SPACING
            for j in range(height_spacing):
                for k in branch_level: print(k, end='')
                else: print(I)

            #BRANCH LEVELS
            for j in branch_level: print(j, end='')

            #LEAF
            if isinstance(tree[i], int) or isinstance(tree[i], float):
tree[i] = str(tree[i])
            if isinstance(tree[i], str):
                if i != len(tree)-1:
                    print(T+E+tree[i])
                else:
                    print(L+E+tree[i])

            #BRANCH
            if isinstance(tree[i], tuple) or isinstance(tree[i], set):
tree[i] = list(tree[i])
            if isinstance(tree[i], dict): tree[i] =
list(tree[i].values())
            if isinstance(tree[i], list):
                if i != len(tree)-1:
                    print(T+R, end='')
                    branch_level.append(I+' ')
                else:
                    print(LR+R, end='')
                    branch_level.append('  ')
                print_tree_aux(tree[i], branch_level, style,
height_spacing)

        if branch_level: branch_level.pop()

    ##ARGUMENTS
    #DEFAULT
    if 'style' in kwargs: style = kwargs['style']
    else:                 style = 'line'

    if 'hspace' in kwargs: height_spcacing = kwargs['hspace']
    else:                   height_spcacing = 0

    print("???")
    print()
    print_tree_aux(tree, [], style, height_spcacing)
    print()

#MAIN
```

```
if __name__ == '__main__':
    echo_off()

    my_set = set([1,4,7])
    my_dict = {}
    my_dict['A'] = 'AAA'
    my_dict['B'] = {'C': 'BBB'}
    my_tree = [1, 2, [3.1, '3.2'], ['4.1', '4.2', ['4.3.1', '4.3.2']],
(5, 6), my_dict, my_set]
    print_tree(my_tree, hspace=1)

    for i in range(10):
        print_progress_bar(i, 10, size=50)
```